

ELXSI

System Foundation Guide

THE SYSTEM FOUNDATION GUIDE

First Edition (October 1987)

Manual EPN: 08856

Binder EPN: 08845

ELXSI System 6400

THE SYSTEM FOUNDATION GUIDE

Copyright © 1987 by ELXSI
San Jose, California

FIRST EDITION (OCTOBER 1987)

Marketing No.: D9512

All rights reserved. No part of this document may be stored in a retrieval system, reproduced, transmitted, or translated into any language without the prior written permission of ELXSI.

ELXSI makes no representations or warranties with respect to the contents hereof and specifically disclaims any implied warranties of merchantability or fitness for any particular purpose. Further, ELXSI reserves the right to revise this publication and make changes to its content without being obligated to notify any person of such revisions.

ELXSI, Embos, Gigabus, and System 6400 are registered trademarks of ELXSI.

ENIX and EMS are trademarks of ELXSI.

DEC and VMS are trademarks of Digital Equipment Corp.

LANCE is a trademark of Advanced Micro Devices.

UNIX is a trademark of Bell Laboratories.

Writer: Scott Lowe

Recognition: David Anderson, Dianne Disousa, John Hawkes, Karlina Ott,
Mike Sherman, Don Wise, Doug Zumbiel

ELXSI

2334 Lundy Place
San Jose, California 95131

Telephone: (408) 942-0900
Telex: 172-320

In Asia and Australia

Tata-ELXSI (Pte) Ltd.
Unit 55, Block 2
Science Park Drive
Singapore Science Park
Singapore 0511

Telephone: 011-65-779-3890
Telex: 78635513

TABLE OF CONTENTS

1	THE SYSTEM 6400	
	INTRODUCTION.....	1-1
	Design Goals.....	1-1
	KEY ARCHITECTURAL ATTRIBUTES.....	1-3
	CHARACTERISTICS OF REALTIME APPLICATIONS	1-5
2	ARCHITECTURAL OVERVIEW	
	ARCHITECTURAL OVERVIEW.....	2-1
	PROCESSES ON THE SYSTEM 6400	2-2
	INTRODUCING THE MESSAGE SYSTEM.....	2-3
	Links, Funnels, and Channels.....	2-4
	Message System Performance	2-5
	Bulk I/O	2-6
	I/O Via Messages.....	2-6
	HARDWARE CONCEPTS.....	2-7
	System Components.....	2-7
	The Gigabus.....	2-9
	Central Processing Units.....	2-9
	Process Context Support.....	2-9
	Cache.....	2-10
	The Instruction Set.....	2-11
	The Memory System.....	2-11
	I/O Subsystem	2-12
	Controller Types	2-14
	I/O Process Control	2-14
	The Service Processor	2-15
	SVP Controller	2-16
	SVP Memory	2-16
	SVP Interfaces	2-16
	SVP Operating System	2-17
	I/O CONCEPTS.....	2-17

Driver Functions on the System 6400.....	2-18
Controller-Code.....	2-18
Supervisors	2-18
Access Managers	2-19
ENIX Device Drivers	2-19
Benefits of the I/O Architecture	2-20
Configuring Controllers.....	2-21
Loading Controllers.....	2-21
Accessing Devices.....	2-22
Through an Access Manager.....	2-22
Direct Access to a Controller	2-23
Transferring Data.....	2-24
Via the Message System.....	2-24
Via Direct Memory Access.....	2-24

3 THE SYSTEM FOUNDATION

INTRODUCTION.....	3-1
System Intrinsic	3-3
Reporting Errors	3-4
Realtime Access.....	3-5
PROCESSES.....	3-6
Process Identification.....	3-6
Jobs.....	3-6
The Process Skeleton.....	3-8
The Process Manager.....	3-9
Spawning a Process.....	3-10
The Process Fork.....	3-11
Creating a UNIX Process	3-11
Lifeline Interrupt Handler	3-12
SCHEDULING AND CPU MANAGEMENT.....	3-15
The Register Set Manager	3-15
Locking a Process in a Register Set.....	3-16
Global and Local Priorities.....	3-17
Modifying the Realtime Priority.....	3-18
Super-Realtime Priority.....	3-19
Launching a Super-Realtime Process	3-20

Process Migration	3-21
CPU Classes	3-22
Default CPU Classes	3-22
CPU Class Queue Restrictions.....	3-23
Restricting Users to a Particular CPU Class.....	3-24
Defining CPU Classes Cooperatively.....	3-27
CPU FailSoft Recovery.....	3-29
Improving the Accuracy of the Interval Timer	3-31

4 MEMORY MANAGEMENT

INTRODUCTION.....	4-1
VIRTUAL MEMORY.....	4-2
Virtual Memory Under UNIX.....	4-4
P0	4-4
P1	4-4
P2	4-5
PAGE MAPS.....	4-8
THE VIRTUAL ADDRESS	4-9
PAGE FAULTS	4-13
Shadow Files.....	4-14
Keeping Data in Memory	4-15
CACHE CONSIDERATIONS.....	4-16
Traditional Cache Schemes.....	4-18
The Cache on the System 6400	4-18
Cachable and Noncachable Data	4-19
SHARING MEMORY.....	4-20
Public-Shared Memory	4-21
Private-Shared Memory	4-22
The \$ShareMemory Intrinsic.....	4-22
The MM\$ Intrinsics	4-23
Fork-Shared Memory.....	4-24
Sharing Memory Under UNIX.....	4-26
DATAPOOLES.....	4-28
Datapool Assembler.....	4-28
Datapool Intrinsics	4-29

Language Interface (Pascal, FORTRAN, C).....	4-30
Programming Examples	4-30
Sharing Datapools	4-32
Cachability and Flushing Registers.....	4-33
Initialized Data	4-33
Case Sensitivity	4-34
Name Scoping Datapool Variables	4-34
Section Names.....	4-34
Naming Convention.....	4-35
MAPPED-FILE ACCESS.....	4-48
Raw-Mapped Files.....	4-50

5 THE MESSAGE SYSTEM

INTRODUCTION.....	5-1
LINKS, FUNNELS, AND CHANNELS	5-1
Creating and Deleting Funnels.....	5-3
Creating and Deleting Links.....	5-4
SENDING MESSAGES	5-5
Send Routines	5-8
NOTIFICATIONS AND RIGHTS	5-10
RECEIVING MESSAGES.....	5-11
Input Fields.....	5-13
Output Fields	5-14
DELETING MESSAGES	5-18
INTERRUPTS AND PRIORITIES	5-18
Local Priorities	5-18
An Interrupting Funnel and Channel.....	5-20
Interrupt Routines.....	5-21
Effect of Local Priority on Interrupts.....	5-22
MESSAGE SYSTEM CONSTRAINTS.....	5-22
PERFORMANCE CONSIDERATIONS.....	5-23
PRIMITIVE MESSAGES.....	5-24
PMSG Receptor	5-25
PMSG Delivery	5-25
Flow Control	5-25

PMSGs Sent by Multiple Processes	5-26
INTERPROCESS COMMUNICATION	5-27
Using the Message System	5-29
Using Shared Memory	5-30
Obtaining Links to Other Processes	5-32
Standard Links and Funnels.....	5-32
All Operating Systems.....	5-32
Embos/EMS Only.....	5-33
UNIX Only	5-33
Server Rendezvous.....	5-34
Example Strategies.....	5-35
Parent and Child Communication.....	5-35
Structuring a General Server Process	5-36
Single-Threaded Servers.....	5-36
Multi-Threaded Servers.....	5-41
Requesting a Service From a Server	5-42
Writing an Interrupt Routine	5-43
USING PASCAL.....	5-44
USING FORTRAN.....	5-45
USING EMBOS C.....	5-48
USING UNIX C	5-50

6 I/O MANAGEMENT

INTRODUCTION.....	6-1
MAJOR ELEMENTS	6-2
Input/Output Processor.....	6-2
I/O Sub-bus	6-3
Parallel Interface Controller	6-4
The Ethernet Controller.....	6-6
VME Front End.....	6-7
High-Speed Device	6-7
Custom Controllers.....	6-9
Communications Supervisor	6-9
The Memory Manager.....	6-9
Realtime Access Manager.....	6-10

Parallel Interface Access Manager.....	6-10
NameSpace Manager.....	6-11
DOWNLOADING FROM THE SVP.....	6-12
CONFIGURING DEVICES.....	6-13
Restricting Access to a Device.....	6-14
Configuration Example	6-15
GAINING ACCESS TO A DEVICE.....	6-16
Access Through an Access Manager.....	6-16
Direct Access to a Controller	6-17
Cleaning Up.....	6-17
TRANSFERRING DATA.....	6-18
A Write Operation.....	6-18
A Read Operation.....	6-19
Memory Organization.....	6-19
The I/O Page Map	6-20
Setting Up for I/O	6-21
Transferring the Data.....	6-21
THE REALTIME ACCESS MANAGER	6-22
Configuring RTAM.....	6-22
Access Via File System Intrinsic	6-22
Parameters.....	6-23
Fortran Example for RTAM.....	6-25
THE PARALLEL INTERFACE ACCESS MANAGER.....	6-28
Configuring PIAM.....	6-28
The ConfigPICdevice Command	6-28
Access Via File System Intrinsic	6-29
Pascal Example for the PIC.....	6-32

7 PARALLEL PROCESSING

INTRODUCTION.....	7-1
THE INTRINSICS.....	7-3
MT\$ShareMemory.....	7-3
MT\$SetupSemaphore.....	7-9
MT\$SetupSemaphoreAndInit	7-10
MT\$SetupLock.....	7-11

MT\$SetupTasks.....	7-11
MT\$GetChildPD.....	7-17
MT\$ShareFile.....	7-17
MT\$OpenSharedFile.....	7-12
MT\$StartTask	7-12
MT\$WaitOnSemaphore.....	7-13
MT\$SignalSemaphore.....	7-14
MT\$Lock.....	7-14
MT\$Unlock.....	7-14
MT\$FlushMemory	7-15
MT\$FlushMemoryAllSharers.....	7-15
MT\$IsCacheablePage.....	7-15
MT\$IsSharedPage.....	7-15
MT\$WaitOnTask	7-15
MT\$ChildDone.....	7-16
MT\$DestroyTasks.....	7-16
MT\$LockStatus.....	7-16
MT\$SemaphoreStatus	7-17
MT\$PageState	7-17
WRITING PARALLEL APPLICATIONS.....	7-18
Sharable Data.....	7-18
Semaphores.....	7-19
Locks.....	7-20
MT\$StartTask	7-20
MT\$FlushMemory	7-20
PARALLEL PROCESSING PROGRAMS	7-21
An Embos FORTRAN Program.....	7-21
A FORTRAN 77 Program.....	7-32
GLOSSARY.....	G-1
INDEX.....	I-1

LIST OF FIGURES

2-1	Components of the ELXSI System 6400	2-8
2-2	The Input/Output Processor	2-13
4-1	The Virtual Address Space Under Embos and EMS.....	4-6
4-2	The Virtual Address Space Under UNIX.....	4-7
5-1	Links, Funnels, and Processes.....	5-2
6-1	The Major Hardware Elements	6-4
6-2	The Parallel Interface Controller.....	6-5
6-3	The Ethernet Controller.....	6-6
6-4	The High Speed Device Using the VME.....	6-8
6-5	The Application, PIAM, and PIC.....	6-31

LIST OF TABLES

5-1	The Receive Intrinsic	5-12
5-2	The ReceiveParamType Parameter Block.....	5-15
5-3	The Receive on Channels Parameter Block.....	5-17
5-4	The Message Formats.....	5-38
6-1	RTAM Request Codes	6-24
6-2	Return Parameters.....	6-25

CHAPTER 1

THE SYSTEM 6400

INTRODUCTION

This manual describes the ELXSI System 6400 from the vantage point of the System Foundation. The System Foundation is the network of processes that controls the system resources. Therefore, this manual presents each element of the system, from a process (the basic software element of the system) to the hardware components, I/O devices, and so on. Because the System 6400 is uniquely suited to perform as a high-speed parallel processor running complex aerospace and realtime applications, the realtime features of the system are described whenever appropriate.

The ELXSI System 6400 is a high-performance multiprocessor designed for realtime, timesharing, and batch applications. The system software provides such features as fast interrupt response, rapid process switching, and the ability to preempt the operating system.

The System 6400 has relatively autonomous and symmetrical Functional Units that attach directly to the very high-speed GigabusTM: one to twelve CPUs, one to four I/O Processors (IOPs), one to six Memory Controllers, and a Service Processor (SVP). I/O devices attach to a device controller; device controllers attach to an I/O Processor. The hardware configuration is flexible, and by adding or subtracting hardware elements, the configuration can be tailored to the particular demands of the workload.

The System 6400

DESIGN GOALS

The ELXSI System 6400 incorporates seven design goals:

- A high performance multiuser system that is appropriate for commercial, scientific, and engineering applications.

The system design assumes that the System 6400 is a resource shared by many concurrent users requiring attributes such as protection, security, accounting, and interactive response.

- Graceful growth through the modular expansion of I/O, CPUs, and memory.

The System 6400 can support one to twelve CPUs, 16 Mbytes to 2 gigabytes of main memory, and 16 to 64 Mbytes per second of I/O bandwidth.

- The operating system processes are not dependent on timing or execution order: they may be interrupted at any point by a higher priority process (such as a realtime application).
- A system that allows nonparallel programs to perform well.

Using a 50-nanosecond CPU cycle time and 64-bit internal paths, the System delivers 8 Whetstone MIPS per 6410 CPU and 12 Whetstone MIPS per 6420 CPU.

- A system that is oriented toward the end user.

The System 6400 provides AT&T's UNIX System V and Berkeley UNIX 4.2 BSD operating systems¹ as well as EMS (ELXSI's version of VMS²) and Embos (ELXSI's proprietary message-based operating system).³

- Compatibility of the next generation of CPUs and other main functional units with the existing System 6400.

¹ UNIX is a trademark of Bell Laboratories.

² VMS is a registered trademark of Digital Equipment Corporation.

³ EMS and EMBOS are trademarks of ELXSI.

The principal example of this is that both CPU models, the 6410 and the 6420, can be freely intermixed in the same cabinet.

- A system that supports the standard VME interface, which simplifies connecting new devices.

KEY ARCHITECTURAL ATTRIBUTES

The key attributes of the machine architecture are as follows:

- A tightly coupled, bus-oriented multiprocessor.
- All CPUs and IOPs have equal access to all of main memory.
- All CPUs have private write-back caches.
- All memory references are to virtual addresses. The instruction set does not support physical memory addressing.
- The system software is a set of concurrent and independent processes that runs efficiently on a variable number of CPUs and is capable of dynamically configuring itself to the hardware it finds to be present.
- The system supports multiple independent memory controllers.
- The system supports multiple IOPs, each of which can support multiple device controllers. Up to four IOPs can be connected to the system, providing a high I/O bandwidth of 64 Mbytes per second.
- The independent Service Processor (SVP) downloads microcode to the CPU, IOP, and memory modules, boots the operating system, monitors performance, and performs diagnostic functions via the system bus.
- Communication between processes is accomplished with packets of data called *messages*. (For more information, refer to "Interprocess Communication" in Chapter 5.)

A *process* is a program and its data. In architectural terms, a device controller is also considered a process because device controllers communicate via messages. Each process's data is protected from access by other processes (unless the data is explicitly shared).

The System 6400

The system architecture supports 256 global priorities, which are classified into four execution categories: realtime, timesharing, batch, and background. Because the operating system modules are interruptible by a higher priority process, a realtime application can preempt the operating system, including the UNIX kernel, without fear of corrupting it. (For more information on this subject, see "Global and Local Priorities" in Chapter 3.)

The System 6400 provides rapid context switching. Each CPU has sixteen sets of registers that define the context of the active processes assigned to the CPU. Switching between processes takes place in as little as 8 microseconds.

Switching contexts can be described as a three-part process:

1. An interrupt is detected by a device controller.
2. The interrupt is passed to the interrupt handler routine (which resides on the CPU) via a message. Message queueing eliminates the possibility of lost interrupts.
3. The CPU switches to the context of the interrupt handler and executes it.

The register set mechanism is ideal for scheduling frames for realtime processing. For realtime operations, register sets can be allocated to critical realtime processes, avoiding any system software involvement in switching contexts and thereby obtaining optimal switching times. (See "The Register Set Manager" in Chapter 3.)

Each CPU contains an interval timer that has a resolution of 50 nanoseconds. For multi-CPU applications in which synchronization is desired, one CPU can be designated as the master; a process on that CPU sends out wake up messages to start the frame cycles of all the CPUs. In this way, synchronization between CPUs can be guaranteed. (See "Timer Services" in Chapter 3.)

The time required to pass a message depends on the size of the message and the priority of the receiving process. To send a message from one process to another, including the process switch and receipt by the second process, takes approximately 115 microseconds, plus .45 mi-

croseconds per byte. Most operating system messages are control and synchronization messages and so tend to be short. Response to an external interrupt, which is the time from the generation of the signal outside the computer to the execution of the first instruction in the software process, is slightly longer, at around 150 microseconds.

Certain external interrupts do not require CPU intervention. In many cases, an intelligent device controller can respond directly to the interrupt. All System 6400 device controllers are intelligent microprocessor-based units that provide direct response to external signals. For example, the System 6400 disk controller can search a disk, locate the desired sector in a file, transfer data, and correct errors without CPU intervention.

CHARACTERISTICS OF REALTIME APPLICATIONS

Most realtime applications are simulations of real world processes, where the computations are based on measurements from outside the system; the results are then fed back into the external process. Realtime programs must complete a cycle (or "frame") within a fixed time that is determined by the nature of the external process (for example, a testbed for aircraft subsystems in which one or more real subsystems are challenged by simulated aircraft). Frame times commonly run between 1 and 10 milliseconds.

Simulations are approximations of a real process that is being modeled. In some applications, an occasional overrun of one frame is acceptable, but in others, a missed frame is not allowed. Because it may be very difficult to recover from a missed frame, realtime programmers must strike a balance between adding code to make the simulation more realistic and missing frames. Programmers rely on guarantees from the computer vendor to make these decisions.

When enhancements to an application put it beyond acceptable frame times, the System 6400 offers the option of increasing computing power by adding CPUs (rather than having to upgrade the entire system).

The System 6400

The combination of extremely fast context switching, message-based interrupts, and direct interrupt response means that, when employed on realtime projects, the System 6400 operating system overhead is very low. The amount of operating system time devoted to context switching and other housekeeping tasks depends upon the application being run. However, based upon benchmark times and calculations, the operating system overhead is less than 5 percent of CPU processing time when the computer is performing realtime tasks.

Many realtime systems require that the system be switched to a special operating system while realtime applications are being run. On the System 6400, engineering productivity can be kept high because the Realtime operating system is the standard operating system. As a result, modern software engineering tools such as full-screen text editors, source code control, and interactive response are available during realtime operations.

CHAPTER 2

ARCHITECTURAL OVERVIEW

This chapter presents an overview of the architecture of the System 6400. Two central notions will be described first: processes and the message system. Finally, we will examine the System 6400's hardware concepts and I/O concepts.

The ELXSI System 6400 supports up to 12 CPUs, 16 Mbytes to 2 gigabytes of main memory, and 16 Mbytes to 64 Mbytes per second of I/O. All of the main functional units, CPUs, I/O Processors (IOPs), and Memory Controllers attach directly to the Gigabus.

The instruction set architecture is a fairly standard scalar architecture: all processes have a 32-bit address space available; all addresses are virtual. There are no instructions for addressing physical addresses. All virtual memory is byte-addressable. Floating point adheres to the IEEE standard.

Word, cache block, and page boundaries are not visible at the instruction set level. Words and registers are 64 bits, cache blocks are 32 bytes (four words), and pages are 2,048 bytes.

In addition to the standard instruction set, there are about forty operating system functions at the instruction level. Most of these functions are message system instructions, but there are also instructions to read the timers, cause breakpoints, read page map entries, flush the cache, and so forth. Certain operating system functions, such as scheduling within a single CPU, reside in microcode.

PROCESSES ON THE SYSTEM 6400

At the center of the system architecture is the notion of a *process*. It is the basic building block of the system. A process is an executing program, including its code, data, registers, and allocated system resources, such as memory. The operating system is constructed out of a network of independently scheduled but cooperating resource managers, each of which is a process.

A process is created each time a program is run. Each process is assigned a unique process ID that is recognizable across all CPUs, a CPU on which to execute initially, a priority, virtual memory space, and several control blocks that manage the process and its messages. This collection of information about a process is called a *process context*. A process context is maintained in the Process Control Block (PCB).¹ Each CPU can store the contexts of fourteen processes simultaneously,² which allows a CPU to switch rapidly between processes while executing one process at a time.

In a multiprocess application, processes may be created programmatically. System processes (that is, processes that are part of the System Foundation) are configured by Elcon and begin to execute when the system is booted. Elcon is the ELXSI system configurator program. Elcon takes three elements as input: a description of the hardware configuration, a description of the software processes that make up the System Foundation, and the configuration profile. Its output consists of a System image and an initial bootstrap file that resides on the Service Processor (SVP). (The SVP is discussed later in this chapter.)

Device controllers are also considered to be processes because they have the same control blocks for process and message management

-
- 1 Other tables define the links and funnels for the process. For more information, see Chapter 5, "The Message System."
 - 2 There are sixteen register sets per CPU, but one is reserved for the Register Set Manager and one is reserved for the CPU itself. For more information, refer to "Scheduling and CPU Management" in Chapter 3.

that software processes do. The IOP to which a device controller is attached manages these control blocks in much the same way that the CPU for a software process does. The definition of device controllers as processes allows them to interact with software processes via messages and makes it relatively straightforward to emulate a controller with a software process. This can be very helpful in developing and debugging complex realtime applications because it allows software development to occur in parallel with hardware development.

INTRODUCING THE MESSAGE SYSTEM

The message system is the principal means by which processes synchronize and communicate. A message consists of up to 888 bytes of data plus some control information. The multiprocessor nature of the hardware is masked by the message system. You do not have to know what CPU a process is running on to send a message to it.

The message-oriented approach to the operating system supports data abstraction with independent processes; that is, makes synchronization between processes and passing data both explicit and visible. The operating system is then transformed into a network of object-manager processes.

Practically speaking, this removes the traditional notion of critical regions from the operating system. To use data communications terminology, the processes on the System 6400 are interconnected via virtual circuits. That is, an explicit point-to-point connection must be established between two processes. But once the connection is established, many messages may be transmitted during the life of the connection.

You may never need to use the message system directly. Processes communicate quite well through pipes using standard File System Intrinsic³ in Embos or system calls in UNIX.⁴ However, you can take

3 For more information on File System Intrinsic, see "System Intrinsic" in Chapter 3, and the *Embos Programmer's Reference Manual*, Vol 1.

4 For more information on UNIX System calls, see Section 2 in the *System V or BSD Programmer's Reference Manual*.

Architectural Overview

advantage of the message system when you need the extra performance it provides or when you are writing a system or low-level service, such as a server or access manager.

LINKS, FUNNELS, AND CHANNELS

The message system consists of machine instructions that operate on *links*, *funnels*, and *channels*. Before a message can be sent from one process to another, a connection known as a link must first be established. A process sends messages on links and receives them on funnels. A process may divide its funnels into fifteen groups known as *channels*. The number of a channel determines its local priority, with 0 being the highest and 15 (the default channel) the lowest. The local priority schedules the order in which messages and interrupts are received, while a process's global priority is used for CPU scheduling.

Typically, a process performs the following procedure:

1. Creates a funnel.
2. Creates a link that points into that funnel.
3. Copies or passes this new link to another process. The second process now has a link to the creating process.

Each link provides a one-way communication path from the sending process to the receiving process. In cases where two-way communication is required, a pair of links is used, with each process having a link to the other process. A link points into only one funnel, but a single funnel can have many links pointing into it. A *link code* identifies the message and is associated with each link. A link code is an arbitrary number defined by the receiving process when it creates the link.

The links for a process are kept in its link table; the funnels for a process are kept in its funnel table. Link IDs and funnel IDs are indexes that select a particular entry in the respective table. A link table entry, for example, contains information such as the link ID, process ID, and funnel number into which the link points. These tables cannot be modified or accessed directly, but by using special machine instructions a process can read individual entries in its link and funnel tables.

Each funnel maintains a FIFO queue of messages sent to its links, and is identified by its Funnel ID. The receiving process may choose to receive a message on a particular funnel or on a group of its funnels. When a message is received, the Process ID of the sending process, and the link code and Funnel ID of the link are received as part of the message control information. This allows the receiver of a message to identify the message. Most links point into funnels in other processes, and are used to send messages to those processes. Each link points to exactly one process.

MESSAGE SYSTEM PERFORMANCE

The message system is fast compared to many software-controlled interprocess communication methods, but slow compared to simple machine instructions. To send a message from one process to another residing on different CPUs, including the process switch and the receipt by the second process, takes approximately 100 microseconds plus an additional .45 microseconds per byte of message data. Sending a message from one process to another process residing on the same CPU takes approximately 130 microseconds (plus an additional .45 microseconds per byte of message data).

The peak data transfer rate through the message system is about 2 Mbytes per second. Bulk I/O, such as from tape or disk, is accomplished by the I/O controllers accessing memory directly into the virtual address space of the process that is reading or writing the data. In this case, messages are only used to synchronize access to the data.

Most operating system messages are control and synchronization messages and so tend to be short. A typical workload consists of brief control messages and significant amounts of work between messages.

Bulk I/O

Because the effective transfer rate of the message system is too slow for moving large volumes of data, passing data through the message system is only useful under special circumstances. Instead, bulk I/O is done by the paging mechanism for disk access and by Direct Memory Access

Architectural Overview

for other high-speed devices. (For more information, see Chapter 6, "I/O Management.")

Disk I/O performance is enhanced by file-system routines (or applications) that tell the Memory Manager or the UNIX kernel to pre-read the pages about to be referenced. Most disk files are opened either for shared-read or exclusive-write access. In those cases, the files can be mapped directly into the user process's address space, so that the file-system routines (or the application itself) manipulate the data.⁵

For devices such as tapes or attached array processors, there is an access manager, but the data is transferred directly into the user's address space. Requests to the access manager include control information and the address where the data is to be read from or written to. The data transfer proceeds directly; when the data transfer is complete, the access manager sends the user process a reply message.

I/O Via Messages

There are times when data is transferred directly in messages. This occurs when 1) you need shared-write access to a file, 2) the amount of data to be transferred is small, or 3) the data is processed between the file and the user process.

An access manager called the Interactive Access Manager (IAM) passes data in messages because the volume of data is so small and there are often several processes accessing a terminal simultaneously. Another access manager called the Keyed File-Access Manager (KAM) also passes data in messages because it must provide correct file-locking and record-locking response.

Under UNIX, I/O between a user process and the UNIX kernel is normally done via messages. (Although bulk data is transferred by a memory-to-memory copy operation rather than via messages.) A UNIX realtime user can arrange to do either mapped I/O or access manager I/O when necessary.

⁵ Mapped files are currently unavailable through the standard UNIX utilities.

HARDWARE CONCEPTS

The power and compact size of the System 6400 is the result of combining advanced semiconductor technology with the System's unique architecture and operating system. By using emitter-coupled logic (ECL) and large scale integrated (LSI) proprietary gate arrays, the System 6400 can produce 12 million Whetstone instructions per second (WIPS) per 6420 CPU.

The System features up to 4 gigabytes of virtual address space per process. (A gigabyte is 1 billion bytes.) The central memory can be expanded up to 2 gigabytes of main memory. The System 6400's modular expandability allows users to add more processing power in a matter of minutes with minimal disruption of operations.

SYSTEM COMPONENTS

The System 6400 consists of a modular multiprocessor and a set of peripherals configured to customer specifications. The CPU is a 64-bit byte addressable multiprocessor that performs high-speed processing of variable length instructions that manipulate a wide range of data formats. The basic system components include the following elements:

Architectural Overview

- Gigabus
- CPUs
- Memory System
- Input/Output Processors (IOPs)
- Service Processor

Figure 2-1 illustrates the functional components of the system.

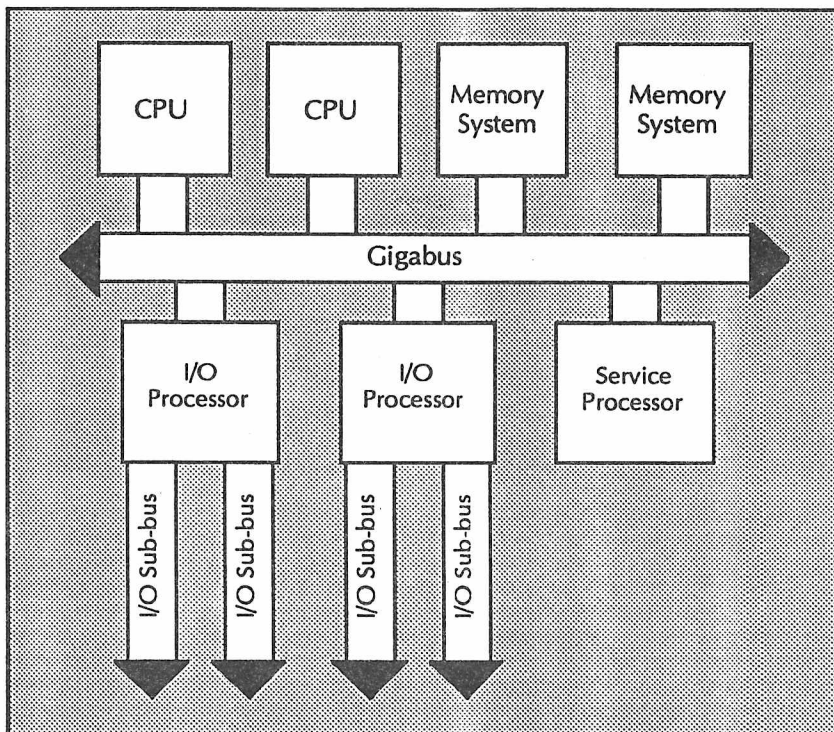


Figure 2-1. Components of the ELXSI System 6400

The Gigabus

The backbone of the System 6400 is the Gigabus. It is a high-speed, synchronous, 64-bit system bus with a bandwidth of 320 Mbytes per second. All functional units of the system communicate via this bus. It accommodates multiple CPUs, memory modules, I/O Processors, and a Service Processor. Usable data rates range from 160 Mbytes per second for Write operations to 213 Mbytes per second for Read operations.

Bus Information Quanta (BIQs) provide the means for communicating between functional units connected to the Gigabus. BIQs are also used to send messages between hardware and software processes. BIQs are composed of small packets (quanta) of information that are transmitted indivisibly between two functional units via the Gigabus. The individual board of each functional unit that sends or receives a BIQ contains the interface logic, which is called the *Gigaport Interface*. The Service Processor, the Cache (on both the CPU and IOP), and the Memory Controller have the Gigaport Interface.

Central Processing Units

The Model 6410 CPU can process up to 8 million Whetstone instructions per second; the Model 6420 can process up to 12 million WIPs.

Process Context Support

In order to keep up to fourteen user processes ready for execution at any time, each System 6400 CPU employs sixteen sets of registers, with one reserved for the CPU and one reserved for the Register Set Manager. The register sets contain the fourteen process states (or contexts) that are maintained in each CPU. The CPU has separate, high-speed memory for the set of registers currently in use.

Architectural Overview

Cache

The cache is high-speed memory that uses a write-back replacement algorithm. Cache is managed with a two-way set associative mechanism that increases the average speed of access to main memory. Each CPU has a cache memory of 16 Kbytes (for the 6410 CPU) or 64 Kbytes (for the 6420 CPU) divided into blocks of 32 bytes.

Each CPU contains 16 translation look-aside buffers (TLBs) that are implemented as two-way set associative caches. Caching is done on virtual addresses. A single TLB stores the virtual-to-physical memory translations for 128 pages of memory. Each TLB contains up to 128 of the most recently written virtual-to-physical address translations for the process with which it is associated.

There are 64 TLB pairs per process. The address of a given page can only use one of a pair of TLB entries. Any two pages that are apart by an integral multiple of 64 pages must share the same TLB pair.

To avoid stale data (that is, having inconsistent values for the same data in two different caches), the System 6400 simply disallows caching of writable shared data. (This can be overridden with the MM\$ intrinsics.) All caches operate independently, as if they were separate computers, leaving it to software to handle any synchronization problems. Since the operating system processes use messages (instead of shared memory) to synchronize and pass data, there are no problems associated with caches operating independently.

A write-back cache can be used (instead of a write-through cache) because the operating system does not need to maintain correct copies of data in main memory nor to put changed data on the bus for other caches to notice. This frees bus bandwidth for more processors by significantly reducing bus traffic.

The Instruction Set

The System 6400 instruction set uses generalized addressing modes and is optimized for code generation by high-level language compilers. A wide range of addressing modes, a complete set of primitives, and variable length instruction formats optimize code compaction and flexibility.

An unusual feature of the instruction set is the absence of privileged instructions or privileged addressing modes. Moving data from one protection domain to another is handled through interprocess message instructions. The message system validates the right of a process to use a particular message path.

The instruction set supports several data types, including Integer, Numeric String (ASCII), String, Logical, Character, and Floating Point. Floating Point is based on the IEEE standard for Binary Floating Point Arithmetic and supports 32, 64, and 80-bit floating point operands.

Data is 64 bits long when in registers, and is of variable length when in memory. Bits in data fields are numbered from left to right, high-order to low-order, with the most significant bit (or byte) as bit (or byte) zero. Each process has sixteen 64-bit general purpose registers. These registers support normal ALU operations and are used for register direct and register indirect addressing. For more information, refer to Chapters 2 and 3 in the *System Architecture* manual.

The Memory System

The System 6400 supports up to 2 gigabytes of real memory. Key features of the memory system include:

- Internal array interleaving
- Detection of single-bit, double-bit, and most multiple-bit errors
- Single-bit error correction
- Memory management that is assisted by firmware and enhanced by cache and translation look-aside buffers (TLBs)

Architectural Overview

- 32-bit addressing, which provides 4 gigabytes of virtual space per user process

The four gigabytes of virtual memory are organized as follows:

- Two gigabytes of private space
- One gigabyte of public space
- One gigabyte of reserved space

Private Space is used for the private data and code of all processes. Each process has two gigabytes of separate private space that cannot be addressed by any other process (except when it is explicitly shared).

Public Space is common to all processes and can be addressed by all processes in the system. A particular byte in the public space has the same address in every process. Public space is read-only. Shared libraries are implemented by installing their code into public space.

Reserved Space is reserved for future expansion.

A general virtual address is a signed, 32-bit integer. Addresses are generally represented by eight-digit hex numbers. Page 0 is unallocatable; if addressed, the process receives an access violation. (For more information on these topics, see Chapter 4, "Memory Management" and Section 12.6 in the *Programmer's Reference Manual*.)

I/O Subsystem

The System 6400 has a large scale I/O capability that allows it to handle tasks that previously required multiple independent mainframes. At the heart of the I/O subsystem are dedicated I/O Processors (IOPs), as shown in Figure 2-2. Up to four IOPs can be configured in a single system. Key performance features of the IOP include a 50 ns processor cycle time, microcoded control of all I/O, and virtual-to-physical memory address translation for transferring data between memory and devices.

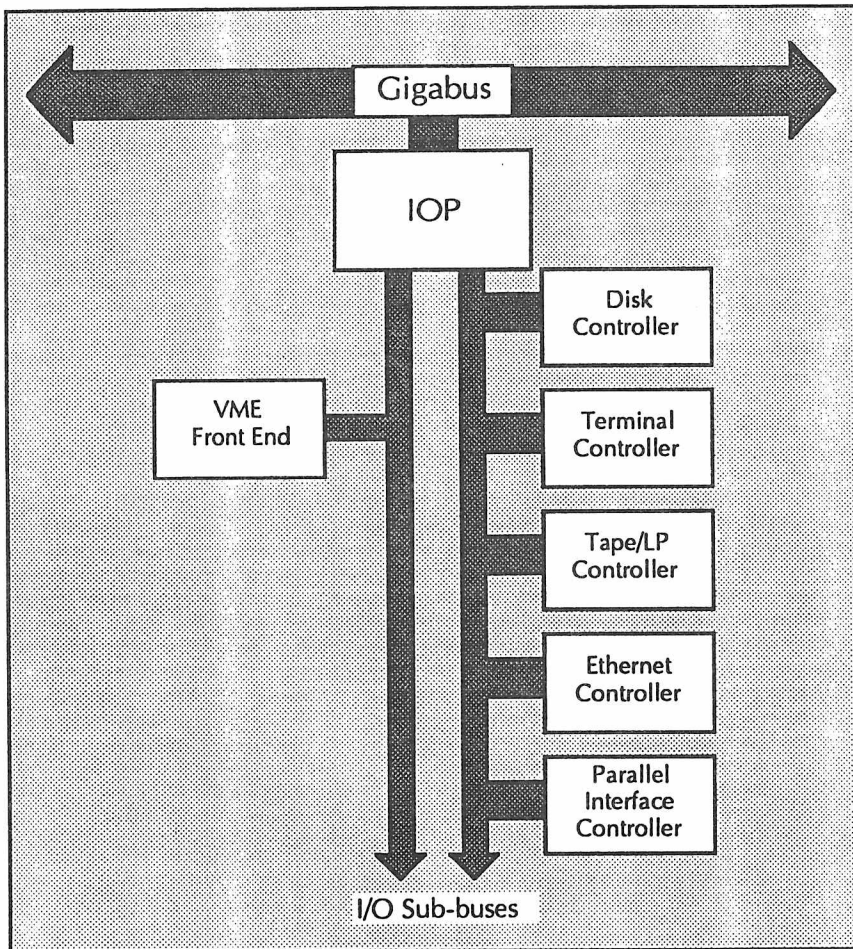


Figure 2-2. The Input/Output Processor

The IOP is similar in structure to the CPU, except where the CPU has a Floating Point Accelerator, the IOP has a Sub-bus Adapter. Because the IOP performs different functions from the CPU, it executes a dif-

Architectural Overview

ferent set of microcode. The IOP handles messages to and from the device controllers and manages the controllers' use of the I/O sub-bus.

All I/O processing is handled by the IOP, freeing the CPUs for computational tasks. Each IOP can have two I/O sub-buses, each of which can operate at 8 Mbytes per second, giving a total bandwidth of 16 megabytes per second per IOP. Each I/O sub-bus can support up to sixteen device controllers.

Each System 6400 device controller contains one or more microprocessors or bit-slice computers. On the System 6400, functions that are usually assigned to the CPU on other systems, such as interrupt handling, seek optimization, and rotational position sensing, are performed by device controllers. (For more information on this subject, see Chapter 6, "I/O Management.")

Controller Types

The controller types that are currently supported include the following:

- a disk controller
- a combined tape/printer controller
- an RS-232/422 terminal controller
- a DEC DR11-compatible interface
- an Ethernet[®] controller
- a sub-bus to a VME front end (VFE)

An IOP can function with any mix of controller types. All controllers are self-identifying and allow dynamic configuration of the I/O subsystem. Each controller has an ample amount of local memory for code, buffering, and tables.

I/O Process Control

Because the various operating system processes communicate via messages to the IOP, the IOP provides a message handler to relieve the individual controllers from having to support the message system. To the System Foundation, each controller appears to be a process. Thus,

like software processes, the I/O processes have Process Control Blocks. Each IOP is capable of managing up to thirty-two concurrent I/O processes (two I/O sub-buses per IOP, and sixteen controllers per sub-bus).

The Service Processor

The Service Processor (SVP) is an independent microcomputer system that attaches directly to the Gigabus. It is equipped with a 20 Mbyte hard disk, a diskette drive, a modem and multiple terminal ports. The SVP starts up the primary system, monitors the hardware, logs error conditions, and diagnoses failed hardware components. The SVP Operating System (SVPOS) not only provides the specialized commands needed for the operations described above, but it also includes a complete set of commands for file editing, copying, deleting, renaming, comparing, and other operations. The modem is used by ELXSI Field Support personnel for remote diagnosis of system failures.

The SVP monitors the performance of the hardware and runs the system diagnostics. It performs bring-up tests when the system is initialized, loads the instruction microcode, and then bootstraps the operating system. To provide diagnostic information, it interrogates the various modules connected to the Gigabus. The SVP monitors the power supplies, system temperature, and cooling fans for any abnormal condition; if such a condition occurs, the SVP shuts down the system before damage can result.

One of the files maintained on the SVP hard disk is the system logfile. Any abnormal conditions detected by the SVP while monitoring or diagnosing the system are noted in this logfile. In addition, the System Foundation processes log error conditions and other system events into the same logfile. During each start-up of the primary system, completion of each step is logged to document its progress.

The major components of the Service Processor are as follows:

- Microcomputer controller
- SVP memory

Architectural Overview

- SVP interfaces
- System clock
- SVP operating system (SVPOS)
- Command port for SVP commands
- Modem port for remote diagnostics
- 20 Mbyte hard disk
- Diskette drive

SVP Controller

The SVP microcomputer controls and monitors all SVP functions. It handles status checks, error checks, interrupts, the system Reset switch and bus control for the SVP. In addition, it checks for errors in the system buses.

SVP Memory

The Service Processor has both EPROM and dynamic RAM storage. The EPROM contains the code that brings up the SVP (by downloading data from a floppy disk). It also contains self-test and diagnostic software for the microcomputer, memory, and floppy disk interface sections of the SVP. The SVP software is stored in 256 Kbytes of RAM, which includes the error correction code (ECC); the ECC performs single-bit error correction and double-bit error detection.

SVP Interfaces

Three interface sections connect the SVP to the System 6400. They are as follows:

- *The Gigabus interface.* The primary means of communication between the SVP and the other system components.
- *The bus control unit interface.* Detects errors on the Gigabus.
- *The power control card.* Monitors the environmental and power supply conditions. Provides 1) power to the SVP, 2)

Architectural Overview

power sequencing to the system, and 3) the SVP interface to the front panel.

SVP Operating System

The SVP operating system (SVPOS) is a general purpose operating system that is stored on the SVP hard disk and loaded into SVP Memory. SVPOS controls diagnostic testing and interrogation of each hardware functional unit. It is controlled from the SVP terminal.

SVPOS consists of a message-based Executive and several permanently resident "system processes," including a file system, a command interpreter, floppy disk and terminal handlers, and two levels of interface to the primary system. The file system manages all SVP I/O devices and disk files. The low-level interface to the primary system allows SVP software to access the Gigabus and its functional units. This interface gives the SVP complete control of all the functional units, including loading microcode and memory, access to functional unit registers, message transactions with processes running on CPUs, and so forth.

The high-level interface, which is active only after the primary system is bootstrapped, allows more sophisticated interaction between the SVP and the primary system. Some examples of high-level operations include logging system events, transferring files to and from the main system, and debugging system processes.

I/O CONCEPTS

The System 6400 I/O system follows the model of using multiple independent processes, each of which has a distinct purpose. For every class of devices, there is a supervisor, one or more access managers, and one or more controllers. For purposes of the message system, the software that runs on a controller is a process. The following description of I/O operations is a model--for each class of devices, the actual implementation diverges from this model somewhat.

Architectural Overview

DRIVER FUNCTIONS ON THE SYSTEM 6400

On most systems, drivers handle request initiation, queuing, and interrupts. On the System 6400, these responsibilities are managed by controller-code running in intelligent device controllers with the aid of the supervisor for the device, along with an access manager or UNIX device driver.

CONTROLLER-CODE

The code running on the System's intelligent controllers manages the functions that on other systems are handled by a device driver; for example, controlling the lines to a device, managing all the low-level interrupts from a device, error-detection, timing requirements, and so on. Controller-code is written specifically for the type of device it manages.

Controllers communicate with software via messages. The format of these messages is defined by the code running in the controller (in agreement with the software process making the request). For message processing, the IOP performs the functions of the CPU for the controller processes. The controllers have their own memory, which stores programs and buffers the data from the device. Data can be moved between a controller's memory and main memory either via messages or by direct memory access (DMA).

SUPERVISORS

Supervisors are processes responsible for the configuration, initialization, and error handling of device controllers. Supervisors are part of the bootstrap image created by Elcon. Since supervisors hold all the links to the controller processes, they control all access to controllers. In order to manage access to devices, supervisors spawn device access managers. As links to a controller are passed to the appropriate access manager or UNIX driver, the access manager or driver can access the devices connected to that controller.

Once an access manager is spawned or the required links are given to a UNIX driver, the supervisor's role is greatly reduced. Supervisors are not involved in routine device operations. After configuration, they normally do not come into play unless a serious problem develops between an access manager (or driver) and a controller process. In these cases, the controller usually sends a message to its supervisor reporting the particular failure. The supervisor generally logs the event in the SVP Logfile. In addition, the supervisor may attempt retries, notify the access manager or UNIX driver involved in the failed operation, or employ other recovery techniques.

ACCESS MANAGERS

The major purpose of an access manager is to control the allocation and sharing of devices under Embos or EMS, and to manage the flow of input and output to devices. Access managers provide a uniform interface for operations such as Open, Close, Read, and Write via the file system intrinsics.

Depending on the particular device, access managers may also manage some of what would normally be considered driver functions. Access managers deal with and monitor the paths to devices, handle data transfer to and from a device, provide access to other system services, and serve as the interface between the user's application and the controller itself.

Programs (especially realtime programs) are not restricted to accessing a device through an access manager. Access managers provide convenient services that can be used if appropriate or bypassed if necessary.

UNIX DEVICE DRIVERS

The implementation of the System Foundation provides UNIX with the unique ability of being able to write standard UNIX device drivers in several different ways. One possibility is for the printer driver to communicate directly with a controller at the lowest possible level--the message system--to achieve absolute and exclusive control over a

Architectural Overview

device. The ENIX disk driver is an example of a driver that uses this technique. It requires the ability to perform transactions with the disk according to the UNIX style of disk management.

Another approach is for a device driver to use an access manager. In this way, the device may be shared by ENIX and other operating systems that may also be running on the same machine. In addition, this approach places the complex task of interfacing with a controller in a common program (that is, the access manager) with a common interface, thereby simplifying the job of writing the device driver. The ENIX device driver is an example of this. This driver uses the Line Printer access manager (LPAM) and the Embos spooler for queuing ENIX print jobs being printed from Embos, EMS, or other ENIX systems.

An ENIX device driver is quite typical of UNIX device drivers in general. The part of the driver that interfaces with the UNIX kernel is standard. The device specific portions of the ENIX driver differ from drivers on other systems in two ways: the way they obtain access to the device (on the System 6400, it obtains a link to an access manager or obtains a link from a supervisor to the controller), and the way they perform the actual I/O (which is via the link). In all other respects, the ENIX drivers adhere to the formal structure required of standard UNIX device drivers.

BENEFITS OF THE I/O ARCHITECTURE

Several important objectives are served by the architecture of the I/O system. First, the System 6400 can perform a great deal of I/O work in parallel. The system's ability to have multiple access managers, multiple I/O transfers proceeding at the same time, and multiple controllers, each managing its own I/O operations, allows I/O to and from devices to be performed in parallel.

A new access manager can be written for each new access method rather than having to add functionality to a single monolithic file system. This allows the access manager code to be customized for a particular device, resulting in faster implementation, better performance, and greater reliability.

Customized access manager code also allows multiple processes to service multiple devices. Because the servers can be spread across the available CPUs, multiprocessor performance is realized. The access managers do not have to communicate with each other because the controller-specific housekeeping code is in the supervisors.

Finally, this partitioning makes it simpler for sophisticated users to construct custom access managers and custom controller-code. Access managers can be spawned processes that can be aborted and restarted without having to reboot the system. Access managers are reasonably standard processes that can be developed in a time-sharing environment.

Being able to use a custom access manager is not a function of a Privileged mode or Super-user status, but rather a consequence of the user's access rights to the appropriate server nodes. The links that a custom access manager holds are specific to the controller and cannot be used to corrupt other controllers or system resources.

Controller-code for most controllers can be developed on the system and downloaded into the controller without booting the system.

CONFIGURING CONTROLLERS

A controller must be installed and configured into the system before it can be accessed. The hardware file on the SVP specifies the type and revision level of each controller, and the IOP and Sub-bus to which it is connected. This file must be updated by adding a line that describes the new controller. In addition, a similar hardware file used by the Elcon system configurator program must be updated; then the system must be rebuilt and rebooted.

LOADING CONTROLLERS

Once the controller is configured and installed into the system and a device is connected to the controller, the next step is to download the controller-code to the controller. Controller-code can be downloaded either by the SVP or a supervisor. Controller-code is downloaded to the

Architectural Overview

controllers each time the system is booted. After the system is booted, controller-code can be downloaded again (if necessary). This is especially useful when developing new controller-code.

ACCESSING DEVICES

Before devices can be accessed, they have to be configured. Configuration includes selecting an access manager to manage the device and specifying other device-specific information such as the device type or Baud rate. This is done automatically every time the system is booted.

There are two ways devices can be accessed: through an access manager (or UNIX driver) or direct access through a device controller.

Through an Access Manager

Under Embos or EMS, a user process typically sends a request to an access manager. The access manager may have to first coordinate with the Memory Manager to freeze memory before it forwards the request to the controller. The controller acts on the request and replies to the access manager, which in turn replies to the user process when the operation is complete.

When a request is sent to an access manager, the access manager does not always forward the request to the controller. One such example of this is an access manager that buffers data, such as the Line Printer Access Manager (LPAM). When a user process requests that a line be written to the Line Printer, the LPAM accepts the request but may not forward the data immediately because the LPAM waits until it has a large cluster of lines before it forwards the data to the printer.

Access managers play a major role in opening, handling requests for access to a device, and closing devices. The functions described in doing so below are not necessarily sequential. When opening a device, access managers

1. Allocate the device to a user process.
2. Control the sharing of devices (when allowed).
3. Prepare the controller for subsequent operation.

When handling an access request to a device, the access manager

1. Validates the application's request. This has two parts: a) It makes sure that the request is in the proper format, and b) checks to see that the process has the rights to access the device.
2. For transfer via Direct Memory Access, the access manager ensures the virtual memory pages are set up for I/O.
3. Forwards the request to the controller.
4. Handles the reply from the controller.
5. May do error handling based upon that reply.
6. Replies to the user process, giving the final status of the request.

A device can be closed by either a Close request or an Abort. When closing a device, the access manager

1. "Cleans up" the outstanding requests to the controller.
2. Deallocates (that is, deletes the links to) the device.

In the case of an abort, the access manager deallocates the device; any outstanding requests to the controller are also aborted.

Direct Access to a Controller

A second way to access a device is to have direct access to a controller. In this case, the application is logically in the same position as an access manager or UNIX driver, sending messages directly to the controller. Direct access to a controller requires that the application take on the responsibility of freezing memory if data is to be passed via DMA and formatting the access request to the device. Also, the application must be responsible for handling the reply from the controller and any error processing that may be necessary if accessing the device is problematic.

When a user process has direct access to a controller, the procedures can be summarized as follows: When the device is opened, the access manager allocates the device to the requesting process. While the

Architectural Overview

device is open, the access manager holds the links to the device. When the device is closed, it deallocates the device.

TRANSFERRING DATA

There are two ways to move data between a user process (or a process running in the mainframe) and a controller: passing data in a message (usually along with a request code specifying what to do with the data) and moving data via Direct Memory Access. Whether an application is accessing a controller via an access manager or accessing a controller directly, both methods of moving data are available (though not all the access managers support both methods).

UNIX device drivers normally transfer data via DMA into the UNIX kernel's address space via messages or with a direct memory-to-memory Copy operation. This allows a process to use a disk buffer cache, so that disk I/O requests to the UNIX kernel are usually satisfied from kernel memory rather than from actual disk I/O.

Via the Message System

The message system is used when the amount of data is small. For example, terminals use this means of data transfer exclusively: every Write request includes the data to be written. This means of data transfer has the limitation of the speed of the message system, about 2 Mbytes per second (which is not fast enough for bulk I/O).

Via Direct Memory Access

When moving data via Direct Memory Access (DMA), the controller can move data directly between the controller memory and the application's virtual address space. When moving data via Direct Memory Access, only control information flows through the message system. Data moves directly between the requestor's address space and the controller.

Architectural Overview

For example, when reading from the tape under Embos, the user process makes an FS\$Read request, which sends the virtual address that the user process specifies along with the Read request to the access manager. The access manager sets up the specified pages for I/O, communicating with the Memory Manager if necessary, and forwards the Read request to the controller. When the controller receives the Read request, it transfers the data directly to the user process's virtual address space. When the data has been transferred, the controller sends confirmation to the access manager. Finally, the access manager informs the user process of the status of the Read request.

CHAPTER 3

THE SYSTEM FOUNDATION

INTRODUCTION

The traditional notion of an operating system often includes everything from basic resource management to the particular aspects of an operating system that make it unique, such as its user interface and file system services. The System 6400 operating system is organized into two parts: the System Foundation, which is devoted to low-level resource management and is common to all the operating systems; and the "User Services" level, which provides higher-level services and makes each operating system recognizable and unique. This design feature gives the System 6400 the ability to provide multiple operating systems on-line at the same time, all of which use the same low-level resource management services.

The System Foundation is a network of processes, each of which controls one or more system resources, such as CPUs, memory, disk access, other I/O access, and so forth. These processes are independent in that they do not share memory with the other System Foundation processes and communicate only through the message system, which allows the System Foundation processes to be distributed across multiple CPUs (a requirement for excellent multiprocessor performance).

How is this different from traditional systems? Most operating systems are implemented as one large process, with all the parts of the operating system sharing the same address space. Such operating systems are complicated because they have to handle interrupts. As operating system code executes, it must be prepared for the possibility that it could be interrupted. While the system is interrupted, system data

System Foundation

structures may change. The only protection against interrupts that a conventional operating system has is the ability to turn off interrupts for a period of time. The stratagems for dealing with interrupts can be very complex. Because the System Foundation processes do not share memory and synchronize via messages instead of interrupts, that complexity is eliminated.

Using the services of the System Foundation, the higher levels of an operating system can provide a consistent interface to system resources that is removed one level from the hardware. This implements the concept of a virtual machine. For example, EMS, Embos, and both versions of UNIX can run simultaneously because they share the same System Foundation. The mechanisms that allocate disk space and memory and transfer data to and from I/O devices are all managed by the System Foundation, regardless of which operating system the user sees. For the most part, the higher-level operating systems built on top of the System Foundation do not need to know about multiple CPUs, process scheduling, process migration, and so on, as these highly machine-dependent operations are handled "behind the scenes" by the the System Foundation.

The System Foundation offers low-level services that make it possible to customize the operating system. For example, under UNIX the only System Foundation services provided for file access are those that allocate a large amount of disk space for UNIX to work with and the service that provides UNIX with direct links to disk controllers. A UNIX file system is created as one large System Foundation file and UNIX manages the space within that file in the way that UNIX users are accustomed to having their files managed. Data is transferred via DMA directly from the disk controllers to the UNIX kernel's memory. This operation is under the kernel's control through the links provided by the System Foundation.

The Embos operating system is the one most closely coupled with the System Foundation. Most low-level Embos services map directly with System Foundation services. For example, an Embos file is identical to a System Foundation file, while under UNIX, entire file systems are

placed in a System Foundation file. Consequently, the Embos user interface manipulates the System Foundation resources. This makes it particularly convenient as a booting and debugging environment for the other operating systems. Therefore, the basic Embos services are shipped with every System 6400, even if the primary operating system on the machine is UNIX.

SYSTEM INTRINSICS

The System Intrinsics are a large set of library routines included with every System 6400. Some of these intrinsics provide access to the System Foundation services, while others provide Embos library functions such as sorting, mathematical functions, pattern matching, and so forth. Many of the System Intrinsics shield applications from the details of the message system, while others establish message communication links with the System Foundation processes. The System Intrinsics are Embos intrinsics, but all of them are callable from EMS and many are callable from ENIX. There are intrinsics that implement both synchronous and asynchronous requests.

Once message links are established, other System Intrinsics can be used to access System Foundation services. This is accomplished by 1) taking a set of specified parameters, 2) formatting them into one or more messages, 3) sending each message to the appropriate System Foundation process, 4) receiving the reply message, 5) extracting the useful information from the reply message, and 6) returning it to the user program via return parameters. If desired, user applications can send messages directly rather than through intrinsics, although this is rarely necessary.

While many of the System Intrinsics do format messages, send them, and receive a reply (notably the File System Intrinsics), other System Intrinsics do not. For instance, if a process reads a file that is mapped into its address space (with the File System Intrinsics), no messages are sent. Likewise, the heap management, pattern matching, sorting, and mathematical function intrinsics do not send messages; instead they work on the data in a process's address space.

System Foundation

The System Intrinsic routines that manage the System Foundation file services and implement the Embos file system are collectively referred to as the File System Intrinsic (FSI). The File System Intrinsic routines are system routines in Embos and (in modified form) EMS that provide a uniform programmatic interface for accessing data in both disk files and devices. The most common operations the FSIs perform include Open, Read, Write, Position, Close, Create, and Delete.

By using the System Intrinsic routines, an application can be written for the System 6400 in much the same way as would be done on other systems. When there is a message to be sent to an Embos or System Foundation process requesting a system service, you would normally use an intrinsic to do that. There is usually little or no advantage in using the message system directly to access system services. (There are advantages, though, to using the message system to communicate between user processes.) Most of the services you could request, such as aborting a process, enquiring about process status, and so forth, are not time-critical. For such tasks, the fraction of a millisecond saved in using the message system directly is of no consequence because you have to wait on the reply from the system process anyway. For complete information on the System Intrinsic routines, refer to the *Embos Programmer's Reference Manual*.

REPORTING ERRORS

Information on all serious system problems, such as unrecoverable I/O errors on any device, is written to the *SVP logfile*. The system intrinsic routines normally report errors by returning an error status code. The main logfile is located on the SVP hard disk, so that it can be read even when the rest of the system is not functioning. The SVP Logfile contains detailed information intended primarily for ELXSI support personnel; users may only see simple messages to the effect that an error was detected. By default, all users have Read access to the SVP Logfile from either the session terminal or the SVP command terminal.

In the Embos Namespace, the most recent part of the SVP Logfile is located at */svp/files/logfile*. When */logfile* is filled, it is renamed as *log-*

file.previous and a new */logfile* on the SVP is started. The old *logfile.previous* is concatenated into a file on the System 6400 disk in the */systemfiles* directory. The name of this file is *logfile.yynn*, where *yynn* is the year and month of its creation. Under EMS, the Logfile is located at *root:[svp.files]/logfile*. This file is not directly visible from UNIX, although it may be listed with UNIX's **embos** command. Some system processes such as the Disk Domain Manager (DDM) use files that are on the system disks to log errors.

REALTIME ACCESS

The System Foundation allows realtime programmers to access resources such as controllers and CPUs at a level in which there is little or no software between an application and the hardware. Most time-sharing and batch operating systems attempt to prevent a single user process from dominating the CPU. Most operating systems that support realtime access, however, do allow realtime processes to dominate the CPU when appropriate. This is done by using a system service that increases execution priority to a level in which the process does not suffer quantum faults, preempts even System Foundation processes, and generally sublimates the influence of the System Foundation.

Such a realtime process can access a resource, in this case the CPU, to such an extent that the process reserves that resource for its exclusive use. Similarly, a realtime process can specify that particular virtual pages be kept in main memory so that as long as the process is running, it avoids page faults on those virtual pages. (Of course, users who wish to employ these services must satisfy the security restrictions defined by their installation.)

PROCESSES

PROCESS IDENTIFICATION

Each active process is identified by a number known as its process ID, which is an index into a system table known as the Process Location Table (PLT). When the system is configured, a limit is set on the size of the PLT, which limits the number of processes that can be active at any one time. While this limit can be as high as 65,535 processes, it is usually set to around 2,000 to save table space. After a process dies, its process ID can be reused.

A process ID is unique across all the CPUs and devices on a System 6400. The process ID should not be confused with the UNIX *PID*, which is unique only within a particular UNIX system. UNIX refers to the system-wide process ID as an *EPID* (ELXSI Process ID). Embos and EMS use the EPID as their only process IDs.

Reusing process IDs introduces a potential problem in the message system. When process *N* dies, other processes may still exist that hold message system links to it. When process ID *N* is reused, those processes holding links to the old process *N* would have access to the new process *N*. This problem is eliminated by associating with each process ID a 24-bit version ID that is incremented each time that process ID is reused. The combination of process ID and version ID is unique, and is checked by the message system each time a message is sent, thereby guaranteeing that no message will ever be sent to the wrong process simply due to a reused process ID. However, under UNIX, additional complications can occur when using process skeletons. (See "The Process Skeleton" below.)

Jobs

Individual user processes are grouped under Embos and EMS into *jobs* to simplify process control, security validation, and accounting. A timesharing job is created whenever a user logs in. Both the timeshar-

ing and batch jobs are controlled and queued by an Embos system process called the Timesharing/Batch Monitor (or TSM). All processes within a single job are identified by a common job ID; they usually have the same user ID and account ID, which are used both for accounting and security validation. Using the job ID, all processes in a job can be suspended, aborted, or resumed as a group.

Each job is started when the Timesharing/Batch Monitor spawns a Job Manager process for the job. The Job Manager in turn spawns a Shell process to execute the commands for that job, and in general retains control over all the processes that are subsequently spawned by the Shell. The Job Manager creates additional Shells whenever you press the **Break** key or run a process detached.

EMS creates processes in the same way Embos does. EMS uses the Embos Job Manager and an EMS Shell that executes EMS commands. Each EMS user session (interactive or batch) corresponds to a particular job.

Since Embos boots UNIX, a particular UNIX system is viewed as a single job from Embos or EMS. All system and user processes associated with a single UNIX system exist within the single job created by Embos when that system is booted. After a job is created, the main UNIX kernel process takes over the Job Manager function of creating, destroying, and controlling processes. The individual processes run as many different UNIX user and group IDs. In addition, to interact with Embos or System Foundation services, the various UNIX user processes may run under different Embos user and account IDs.

When using Embos system monitoring tools to view a UNIX system, processes associated with a particular UNIX system may be identified in one of two ways: by their common job ID or by an initial character prefixed to each UNIX process name. This character, which is selected by the system administrator, is different for each UNIX system running on a System 6400.

System Foundation

The Process Skeleton

A process skeleton is a set of all the necessary data structures required to create a process. When a process terminates, its data structures are not automatically deallocated: they may be saved to be used again when the next process is spawned. While the data in the data structures will be different, the data structures themselves may be reused. Because the work of allocating and deallocating the data structures does not have to be repeated for every process, a process skeleton reduces the time it takes to create a process.

UNIX systems on the System 6400 maintain process skeletons composed of complete processes. They can be reactivated very quickly in order to make Fork and Exec operations extremely fast. A process skeleton maintains its EPID across each of its "lives." Its UNIX PID is different each time, however, which follows normal UNIX procedures for PID allocation. After an Exec, a UNIX process may have a different EPID, though it will always have the same PID. You can find the EPID under UNIX with the `ps -l` command.

The System Foundation itself maintains a pool of previously used virtual memory files called *shadow files*. (See "Shadow Files" in Chapter 4.) These files reduce the amount of time it takes to perform a System Foundation spawn or Fork operation. A new EPID is assigned for any System Foundation process that is created, regardless of whether it uses a new shadow file or one taken from the pool.

For UNIX processes that access System Foundation services or use the message system directly, some restrictions regarding the use of skeletons are enforced. These restrictions are necessary to avoid problems with a *dangling link*. A dangling link occurs when a newly activated process skeleton receives a message from a process that had a link to it in its last life, but should not have a link to it now. These restrictions are normally transparent to applications, except for the possibility of slightly degraded performance on Forks and Execs. No such restrictions apply when using pooled shadow files.

THE PROCESS MANAGER

The Process Manager (PM) handles process skeletons (for Embos) and directs the creation of a new process (when a process skeleton cannot be used). It starts with information stored in the program file that is being forked or spawned and allocates the system resources necessary to create a process suitable for executing that program. The Process Manager carries out the following functions:

- Assigns a Process ID (EPID).
- Sets up a shadow file, which is a temporary disk file that holds a process's data when the data is not in memory.
- Creates the control blocks for the process, including a Process Control Block, a Link Table, and Funnel Table with the appropriate set of links and funnels.
- The PM works in conjunction with the Memory Manager, which allocates pages of physical and virtual memory, and the Register Set Manager, which manages the CPU to which the new process is initially assigned.
- Once a process is set up to execute, the Register Set Manager allows the process to execute in the normal priority-ordered manner.
- Finally, it sends a message back to the requestor (which is normally a Job Manager), who replies to the creating process, informing it that the spawn or fork is complete. (See "Spawning a Process" below.)

When the system is being used interactively, most processes are spawned by a Shell. The major purposes of a Shell are as follows:

- Interpret a command.
- Identify the executable file associated with that command.
- Use the Job Manager or UNIX kernel to spawn the executable file into a process.
- Open the standard input and output files for the process.

System Foundation

- Pass the parameter information specified on the command line to the process.
- The Shell waits for the process to complete and then performs any necessary clean-up. The Shell is then free to interpret the next command. A user process may instead instruct the Shell to return immediately after initiating a process. Such processes are called *background* or *detached*.

SPAWNING A PROCESS

The System Foundation supports two methods for creating a new process: *spawn* and *fork*. For both mechanisms, all of the essential components of a process must first be identified and brought together into an executable form. The essential components of a process are the code to be executed, any initial variable values, and an upper limit of the memory space to be used by the process. A *spawn* and *fork* differ in the sources from which these components are gathered. A *spawn* requires an Embos Bound file (or an EMS executable file).

Under Embos, a program called the *Binder* combines this information and specified relocatable libraries into an executable page image file called a Bound file. The first page of the Bound file, called the *Info Page*, contains important parameters, including the location and size of each section of the Bound file. The Binder selects locations for relocatable code and data sections within the virtual address space, modifies the relocatable addresses, resolves external references, and assigns the page protection attributes for each section.

A process is spawned in the following manner:

- First, the process originating the *spawn* (called the *parent process*) gives the Job Manager the name of the Bound file (in a message, of course), along with specific instructions such as the process priority or CPU class.
- The Job Manager passes the request to the NameSpace Manager (NSM).

- The NSM looks up the name of the Bound file, verifies the access rights of the parent, and then passes all the necessary information to set up and execute the new process (called the *child process*) to the Process Manager.
- The Process Manager supervises the collection of whatever system resources are needed to run the process.

THE PROCESS FORK

The second type of process creation is called a *process fork*. When a process spawns, the spawned process can be an arbitrary Bound file. But in the case of a process fork, the parent process creates a copy of itself and both the parent and the child processes execute concurrently. Process forks are the only way process are created in a UNIX environment, and are used in parallel processing and realtime applications as well.

The main advantage of the fork is that it is a faster method of creating a process. Because the parent process already exists, the Bound file and most of the process's information is already in place. The read-write portions of a forked process use a different shadow file because the parent or child process may alter these areas after the process fork takes place, in which case the changed values must not be reflected in the other process. To save memory and copying time, the read-only sections, such as the program code, can be shared between the parent and child.

CREATING A UNIX PROCESS

UNIX has two system calls that create new processes and retrieve the text and data associated with the process: *Fork* and *Exec*. Like a fork in Embos, Fork replicates the entire parent process bit-for-bit. The child process is an exact clone of the parent process, except for the Process ID.

As soon as a child process is forked, it usually executes an *Exec* system call, which requests new text and data for the child process. You can

System Foundation

pass a number of parameters to an Exec call, but since Exec overwrites the process' memory, the parameters must be cached before the process starts running. The parameters for the Exec are copied to the the Arg-Cache, the new process image is loaded, and just before the process starts, the parameters are copied again from the Arg-Cache to the top of the stack.

LIFELINE INTERRUPT HANDLER

The Lifeline Interrupt Handler (LIH) consists of a special message system funnel that is set at the highest priority and an interrupt routine associated with that funnel. The LIH performs basic functions of process control. It cannot be disabled, so it is always possible for the operating system to control a process by sending messages to the LIH, regardless of the status of a process. Under Embos and EMS, most of the messages going to the LIH come from a process's Job Manager, which has links into the LIH funnels of every process in the job. Under UNIX, the UNIX kernel process (EM or BM) holds the links to its process's LIH funnels.

The LIH performs the following functions:

1. Starts a process executing.
2. Dismantles a process and stops any further execution when a process is aborted.
3. Puts a process into the Debugger.
4. Temporarily suspends execution of a process.
5. Resumes execution of a process.
6. When requested, gives the Job Manager or UNIX kernel a process's accumulated CPU time.

NOTE

The first four functions in this list can be requested at unpredictable times as a process executes. The LIH uses the standard interrupt mechanism, which assumes that register 15 points to the top of the stack; that is, it assumes that memory above that point can be written over. To ensure predictable operation, therefore, you should not use register 15 for anything other than the stack pointer.

TERMINATING A PROCESS

A process can be terminated in a number of ways, depending on whether its termination is normal or abnormal and whether it terminates while the process is executing or is forced to terminate from a condition outside the process. A normal termination occurs when a program completes execution by exiting its outer block, or by explicitly calling the `$Exit` system intrinsic under Embos, the `Sys$Exit` intrinsic under EMS, or the `exit(0)` system call under UNIX.

An abnormal termination is caused during execution by an "exception" such as an arithmetic overflow or divide by zero, or by an explicit call to the `$ErrorExit` system intrinsic. When an abnormal termination occurs, the status stack for the process is displayed, showing any errors detected by system intrinsics. For more information on these intrinsics and the status stack, refer to the *Embos Programmer's Reference Manual*.

If a program in a Shell file terminates abnormally, an error message is displayed, and execution of the Shell file is terminated. This can be avoided by executing the program via the **Run** command and specifying the `+continue` switch. Then if the program terminates abnormally, the error message is still displayed, but the Shell file executes the next command.

System Foundation

The JM\$Kill intrinsic allows you to specify a process or set of processes within the current job and indicate whether they are to be *Hardkilled* or *Softkilled*. (See the Help file for more information on this intrinsic.)

The Job Manager or UNIX kernel can terminate any of the processes in its job by sending a termination message into the LIH funnel of the process, where it is received and executed by the LIH. For a Softkill, the LIH attempts to close all open files and take whatever other actions are necessary to clean up the state of the process so that resources allocated to the process are gracefully released. A Softkill is an exception: a process can define its own Softkill handling routine, which is invoked by the LIH when it receives a Softkill message. A Softkill does not guarantee that a process will be terminated, although it usually is. UNIX uses signals to interrupt processes, although it can also execute a Hardkill.

If some files cannot be closed, or if the process is so severely damaged that it cannot complete the execution of a Softkill, a Hardkill message may be necessary. For a Hardkill, the LIH dismantles the process without attempting to release system resources. The process cannot define its own Hardkill handling routine. A Hardkill is a more primitive operation, and it almost always terminates the process, but pending output may not get written back to disk. Opened disk files are closed abruptly and physical memory is released.

The termination of a process is detected primarily through the message system. Any process to which the dying process has a link may be notified of the process's termination by a special message called a DeleteLinkNotification message. This message is automatically sent if the link was created with that notification right. If processes that hold links to the dying process attempt to send messages to that process, they will receive an error status that indicates the receiving process is dead. You can also detect the termination of a process interactively by using the appropriate process status command.

SCHEDULING AND CPU MANAGEMENT

THE REGISTER SET MANAGER

The Register Set Manager (RSM) is a System Foundation process that works closely with the CPU microcode to control the priority and scheduling of processes on each CPU. The RSM also works closely with the Process Manager (PM) to perform the process migration and CPU Failsoft functions. There is one RSM process per CPU.

Each CPU has an *Active List*, which is a priority-ordered list of the processes ready to be executed. The highest priority processes on the Active List are assigned by the RSM to register sets, which are data structures maintained by the CPU that contain or point to all the information needed to execute a process. There are a total of sixteen register sets per CPU: one is reserved for the CPU itself and another for the RSM, leaving fourteen register sets available for general use. The Register Set Manager decides which processes on the Active List should be assigned to register sets.

At any instant, only one process per CPU executes. To determine which one of the processes in the register sets should be executed next, the CPU follows this rule: *the highest priority process that is in a register set and ready to run executes until it is blocked or suspended*. When the executing process is blocked or suspended, the next highest priority process executes.

There are various conditions that can block a process. The most common blocking condition occurs when a process executes a synchronous Receive instruction and no message is waiting. Another type of blocking condition is called a *fault*. There are several types of faults, including page faults, quantum faults, and Copy on Write (COW) faults. A page fault occurs when a page of virtual memory is accessed that must be brought in from disk. A quantum fault occurs when a process has completed its timeslice. The RSM decides how long a timeslice should be, based on the priority of the process and how much CPU time it has used. A timeslice is typically about 50 milliseconds, though

System Foundation

a background priority, CPU-intensive process may be assigned a timeslice as long as 100 milliseconds.

Normally the RSM executes as a low priority process, so it tends to run only when there is nothing more important for the CPU to do. However, when the microcode needs the RSM to make a scheduling decision, the RSM runs as the highest priority process on the CPU. When all other processes assigned to register sets are inactive, the RSM wakes up and decides which process to activate next. The RSM activates a process by assigning it to a register set, which may first require that the RSM remove an inactive process from its register set.

Locking a Process in a Register Set

One of the functions of the Register Set Manager is to select which processes on the Active List are assigned to a register set. There are certain events, such as waiting for receipt of a message, that might cause a process to be taken out of a register set. When it's time for that process to begin executing again, it takes longer to get started because it requires the intervention of the Register Set Manager to get the process back into a register set. That delay can be avoided by "locking" the process into a register set, which means allocating exclusive use of that register set for as long as the process is alive (or until the register set is "unlocked"). Locking a process into a register set is commonly done when a realtime application receives timer interrupts or external interrupts and must guarantee the fastest possible response to those interrupts.

The advantage of locking a process into a register set is that the process is always capable of executing. The disadvantage is that it reduces the number of register sets available for other processes running on that CPU, creating both more competition for the available register sets and more overhead for the Register Set Manager as it attempts to resolve the competition.

GLOBAL AND LOCAL PRIORITIES

Processes can execute at any one of 256 *global priorities*, with Priority 0 being the most preemptive and Priority 255 the least preemptive. Each process can execute at any one of 16 *local priorities*, which correspond to the sixteen possible message system channels. The mapping of a process's local priority to its global priority is managed by the Register Set Manager and specified by the process's Channel Priority Map.

The System Foundation divides the 256 global priorities into four *execution classes*. The highest of these is realtime priority, which is the execution class used by most System Foundation processes. Then in descending order, the execution classes are timesharing, batch, and background. The execution class in which a process executes is determined at run-time. Further, a realtime process under Embos can be assigned its Channel Priority Map prior to run-time with the **ChannelPri** command, which stores this mapping in the Bound file.

For processes executing in the timesharing, batch, and background execution classes, the Register Set Manager (RSM) dynamically adjusts its global priority based on the behavior of the process and overall system load. For example, a process that consistently uses a lot of CPU time will have its priority reduced to a less preemptive level. When the process's CPU usage again becomes normal, the RSM raises the priority of the process. The process stays within its execution class unless it is explicitly changed with the `Pro$ResetPriority` intrinsic.¹

The message system can dynamically change the global priority of a process within an execution class as messages are sent to the process. The funnels of a process are grouped into sixteen channels, each of which can be assigned a different priority level within the execution class. A process's global priority is determined by the priority level of

¹ This intrinsic allows you to specify priority levels for each channel. For more information and some programming examples, refer to the Help file.

System Foundation

the highest priority channel that has a message pending. The priority level of a particular channel is referred to as a "local priority"; for example, the priority level of Channel 9 is called "local priority 9."

The realtime execution class ranges from priority 1 to priority 80, with a default realtime priority of 70. A realtime process executing at a priority in the range from 1 to 9 is known as a *super-realtime process*. At this high global priority level, the process preempts even the System Foundation processes. In addition, quantum faults (which cause processes to be periodically preempted) are disabled. If the process is also locked into a CPU register set, its response time to a received message will be optimal. (See "Super-Realtime Priority" later in this chapter.)

Realtime processes have some control over their global priorities. A realtime process can accept a default Channel Priority Map with global priorities near the inferior end of the realtime priority range. Alternatively, a realtime process can specify a Channel Priority Map with individual global priority values associated with its sixteen channels.

Since the Register Set Manager never modifies a realtime process's global priorities, a compute-bound realtime process will always preempt a process with an inferior global priority. However, prolonged, compute-bound execution at a realtime priority is discouraged. A process designed to function as both a high-response, minimal-latency server and also as a computational server should specify channel priorities in both the realtime and timesharing ranges as described above. By carefully assigning the proper funnels to each channel, a single process can effectively operate in both execution classes.

MODIFYING THE REALTIME PRIORITY

All users have the right to change their local priorities. But changing the global priorities is only relevant for a realtime process that also has the correct security access to the Process Manager rendezvous that supports this service. The file `/systemfiles/groups` identifies which account IDs can spawn realtime processes.

When a process is launched as a default realtime process, the default priority assigned to all sixteen channels is 70 (see “Launching a Super-Realtime Process” below). This places a realtime process at a superior priority than the lower three classes and at an inferior priority than most of the System Foundation processes (however, some System Foundation processes, such as the Disk Domain Managers, also run at Priority 70).

To specify a priority level other than Priority 70, use the **ChannelPri** command² to define precise priority values for any and all channels. That specification is put into the Bound file and referenced by the System Foundation when the process is spawned as a realtime process; the RSM then places those values into the channel priorities.

Another way to modify realtime priorities is the **Pro\$ResetPriority** intrinsic. (See the Help file for more information on this intrinsic.) The Register Set Manager ignores any attempt by a non-realtime process to specify channel priorities.

SUPER-REALTIME PRIORITY

Super-realtime priority is a special instance of realtime priorities. A super-realtime process is defined as a process whose highest priority channel (Channel 0) is set to a priority in the range from 1 to 9. Super-realtime priority is a higher priority than any operating system process. Processes running at that priority can interrupt or preempt any operating system process, which guarantees the fastest possible response to messages or interrupts.

A super-realtime process never blocks on a quantum fault; when the process uses its timeslice, the microcode simply gives it execution time again. Such a process can continue to execute for as long as it remains compute-bound. This process can be preempted by another super-realtime process with a higher priority. If a super-realtime process

² This command sets the channel priority map in the Bound file. These priorities only take effect if the Bound file is spawned as a realtime process. For more information, refer to the Help file.

System Foundation

executes at global priority 1 (which is the highest priority the System Foundation allows) and becomes compute-bound, it will dominate the CPU's resources.

Designers of super-realtime processes do not want them to be taken out of a register set, even if the process is blocked. To guarantee undiminished execution times, these processes can be permanently allocated to (or "locked into") a register set. This is done via an explicit System Foundation call, `Pro$LockInRset`.³ (For a further discussion on this topic, see "Locking a Process in a Register Set" earlier in this chapter.)

Launching a Super-Realtime Process

The following list presents the sequence of tasks to be completed when launching a super-realtime process.

1. Set up the account in the *groups* file so that it can run realtime. (See */embos/documents/groups.doc* and "Restricting Users to Particular CPU Classes" later in this chapter.)
2. If you are using the Timer Manager for timer work, that process should also be locked in a CPU and locked in a register set. For Software Release 12.0 and beyond, this is the default configuration.⁴ (For a related discussion, see "Timer Services" later in this chapter.)
3. Use the **ChannelPri** command to make the process super-realtime so that it will not quantum fault (Channel 0 must have a

³ The `Pro$LockInRset` intrinsic locks a process into a register set within the CPU the process is running on. A process must be locked into a CPU *before* it is locked into a register set. For more information, refer to the Help file on this intrinsic.

⁴ To run a super-realtime process on a system with system software prior to 12.0, the System Administrator should replace the SPM entry in the *softwaretss* file (at */embos/systemconfig*) with the following (where "nn.n" is your software version number):

```
SPM V /Embos.nn.n/bound/spm cpu=cpul +lock
```


priority between 1 and 9) or at run-time call `Pro$ResetPriority`. (See "Global and Local Priorities" earlier in this chapter.)

4. Launch the process as a realtime process with this **Run** command:

```
Run processname pri=realtime
```

5. Lock the process in the CPU by using `Pro$LockinCpu` or the following **Run** command:

```
Run processname pri=realtime CPU=cpuN
```

where "cpuN" is the CPU name.

6. Lock the application process in a register set (with `Pro$LockInRset`).
7. Use `MM$FreezeAndTouchProcess` or `MM$FreezeProcess` to freeze the process in memory so it will not page fault. For more information on these intrinsics, see the Help file and "Keeping Data in Memory" in Chapter 4.
8. If you have not used the **ChannelPri** command to reset global priorities, then you must call `Pro$ResetPriority` to make the process super-realtime.

PROCESS MIGRATION

Process migration is a system service that moves processes from one CPU to another CPU in order to balance the workload across CPUs. It is the mechanism by which the system load can be dynamically adjusted. Periodically, each Register Set Manager sends a message to the Process Manager that contains two items: a list of processes that could be migrated and a numerical value called the *load factor* that describes how busy its CPU is. The Process Manager analyzes these messages to decide if there is a load imbalance. If so, the PM works with the RSM on the busiest CPU and the RSM on the least busy CPU to first select the most appropriate process to be migrated to another CPU and then relocate all of the process's CPU-resident data structures to that CPU.

System Foundation

The Process Location Table entry (which specifies which CPU the migrating process is currently running on) is updated to note the new destination CPU. Any data the process may have in the cache of the source CPU must be flushed to main memory, where it can then be accessed by the destination CPU.

When a process is spawned, the Process Manager assigns a CPU class for that process (which, by default, is `$any`). (See "CPU Classes" below.) A process can then migrate between the CPUs in that CPU class only. If a process is assigned to a CPU class that only has one CPU in it, the Process Manager will not migrate the process. But if a process is assigned to a CPU class with one or more CPUs and it becomes a candidate for migration, the Process Manager will migrate it to another CPU in that class.

CPU CLASSES

A CPU class is a set of one or more CPUs grouped together to bring scheduling and process migration under strict control. Each CPU in the system has a unique name (such as CPU1, CPU2, and so on). Each CPU is itself a CPU class that is assigned the same name as the CPU. Depending on the number of CPUs and how tightly they are to be controlled, the system administrator may define multiple CPU classes. For a process to run on a CPU class, it must have at least one valid CPU. (If not, the **Run** command will fail.)

Default CPU Classes

There are two default CPU classes, `$all` and `$any`. The `$all` CPU class includes all the CPUs on the system, and cannot be changed. The `$any` CPU class includes all of the CPUs on the system that are to be used by default. Normally these two classes are the same, except when one or more CPUs are being reserved for a specific purpose (such as running realtime applications).

There is also a default CPU class for each CPU. When the system is booted, the Process Manager first defines each CPUs as a separate CPU class and then defines additional CPU classes according to the

CPUclass commands in the System Profile (if there are any). At the same time, the Process Manager adds each CPU to \$all and \$any CPU classes.

For the Process Manager to do its work, there must be one CPU class that contains all the functioning CPUs on the system. If one of the CPUs halts, the system removes it from \$all. This is the only case in which \$all can be modified. The CPU class defaults to \$any under the following conditions: 1) when a process's \$CPUclass Shell variable is not defined, 2) the CPU class is not defined in the account statement in the *groups* file, or 3) the CPU class is not set on the command line (with the **Run** or **Batch** commands).

CPU Class Queue Restrictions

You can restrict certain batch queues or certain accounts to run on certain CPU classes (as shown in the example below). Normally there is no restriction. When a user's account is restricted to a particular CPU class, that is the CPU class his or her processes are run on, regardless of any other CPU class specification. If there is no account restriction, but the current job is running from a batch queue that has a restriction, then that restriction is applied. The account restriction is specified in the the account statement in the *groups* file, along with the password and other information about the account.

The CPU class restriction for a batch queue is specified in the System Profile, along with all the other information about that batch queue. Optionally, there is an entry in the System Profile for each batch queue that specifies the type of batch queue, the maximum number of jobs that can run at one time from that queue, and so on. One of the items in the batch queue entry is a CPU class restriction. That restriction is applied to processes run within batch jobs if there is no restriction on the account the job is running under. Lastly, any restriction that the user specifies on the **Run** command, such as

```
Run program CPU=CPU2
```

is applied only if there are no restrictions on the batch queue or account. If you specify in the *groups* file account statement

System Foundation

`CPUclass=$any`

every process the user runs is forced to \$any (that is, the Process Manager will choose which CPUs to run the process on). Any CPU class restriction the user specifies will be ignored. For example, if the user enters

`Run Prog1 CPU=CPU1`

the CPU specification is ignored and Prog1 is forced to \$any, even if the CPU is one of the CPUs in the \$any CPU class. The system does not check which CPUs are in which classes. It simply takes whatever is in the account specification in the *groups* file, or if there is no entry there, it takes the specification for the batch queue in the System Profile. When \$any is specified in the *groups* file account statement, the user is prevented from specifying a particular CPU.

Restricting Users to a Particular CPU Class

CPU class residency can be declared for particular accounts, specific batch queues, or as a **Run** command-line parameter for a particular program. The steps to permanently restrict a group of users under specific accounts to a particular CPU class are described below.

If you want the CPU class definitions to take effect before the next boot, you must use the **CPUclass** command to define them (see Step 1 below). Then, to retain these CPU class definitions, the CPU classes must be defined in the System Profile (as shown in Step 2). (The *systemprofile* file is in the */systemfiles* directory.) In conjunction with this, the *groups* file entry for each user must include the appropriate CPU class entry. (The *groups* file is also in the */systemfiles* directory.)

In our example, the system has five CPUs that must be shared among two departments. Each department has its own CPU class and is allotted two CPUs. CPU5 is reserved for realtime work. A word of caution: While such a configuration satisfies the administrative requirements of an organization, it hampers the efficiency of the system because it restricts the options available for process migration. A chronic imbalance in the CPU workload could result.

System Foundation

1. To have CPU class definitions take effect immediately, create the CPU classes with the **CPUclass** command (see the Help files for detailed information on this command):

```
CPUclass Dept1 CPU1, CPU2
CPUclass Dept2 CPU3, CPU4
CPUclass realtime CPU5
```

The CPU classes defined at the command line remain in effect only until the next boot.

2. The System Profile is read when the system is booted. The CPU classes specified in the System Profile remain in effect as long as the CPU class entries are in the System Profile.

The system administrator makes the following entries in the System Profile:

```
CPUclass classes = Dept1, Dept2, Realtime
CPUclassDept1 subclasses = CPU1, CPU2
CPUclassDept2 subclasses = CPU3, CPU4
CPUclassRealtime subclasses = CPU5
```

Notice that when you assign the CPUs to the subclasses, the name of the CPU class is appended to the **CPUclass** command

3. At the system prompt, the system administrator enters this command:

```
ReopenProfile
```

Now the CPU classes defined in the System Profile will be redefined every time the system is booted.

System Foundation

4. The system administrator makes the following entries for each user in the *groups* file. In our example, James is a normal user of the system and a member of Department 1 (Dept1). Laura is a realtime programmer and a member of Department 2 (Dept2). Notice that in her case, no CPU class restriction is included in her account statement. Thus, James must run his programs on Department 1's CPUs, but Laura can run her programs on any CPU.⁵

```
user james account=james
password=nx03ym9iyblz
account james users=james CPUclass=Dept1
user laura account=laura
password=vqladm7tvwc3
account laura users=laura
```

5. The system administrator enters this command:

```
ReadGroups
```

The users and accounts defined in the *groups* file are redefined immediately without having to reboot the system.

6. Since Laura's account in the *groups* file does not specify a CPU class, her programs would default to the \$any CPU class, which usually includes all the CPUs on the system. By making the following entry in her *logincommand* file, Laura's programs will run by default on the Department 2 CPUs .

```
Set $CPUclass Dept2 declare = global
```

In this way, the strict partitioning of CPU work is maintained (and Laura does not have to specify a CPU class every time she runs a program).

⁵ Passwords are encoded in the *groups* file for security purposes. See the online document "GroupsAndAccounts" for more information.

7. Finally, when Laura does want to run a realtime program, she can do so by specifying the realtime CPU class at the command line, typically with the **Run** or **Batch** commands:

```
Run Prog1 CPU = Realtime Pri = Real
```

An attempt by any other user in either department to run a program on the realtime CPU will fail because their entry in the *groups* file takes precedence.

Defining CPU Classes Cooperatively

The previous example showed how to permanently restrict a group of users to a particular CPU class. This section describes a more cooperative approach to setting up CPU classes on your system. Let us say, for example, that a field analyst wants to create new CPU classes that correspond with the 6410 and 6420 CPUs on the system. He also wants to set it up so he runs on the 6420 CPU class by default. To define the new CPU classes prior to a reboot, the analyst uses the **CPUclass** command from the command line. Then he defines the CPU classes in the System Profile so that they will be created each time the system is booted. For his convenience, he sets the \$CPUclass variable in his *logincommand* file.

1. At the command line, the analyst defines the CPU classes as follows:

```
CPUclass6410 CPU2, CPU3  
CPUclass6420 CPU1, CPU4
```

This sets up the desired CPU classes initially, but they will remain in place only until the next boot.

2. In the System Profile (*/systemfiles/systemprofile*), the analyst enters the following:

```
CPUclass classes = 6410, 6420  
CPUclass6410 subclasses = CPU2, CPU3  
CPUclass6420 subclasses = CPU1, CPU4
```


System Foundation

3. The system analyst issues the **ReopenProfile** command. Each time the system is booted, the new CPU classes are created.
4. To define a default CPU class other than the system default (which is \$any), set the \$CPUclass Shell variable. The analyst makes this entry in his *logincommand* file:

```
Set $CPUclass 6420 declare = global
```

The first time he logs in, the CPU class default is put into effect. Thus, any processes spawned in that job will run on the specified default CPU class (which in this case is the 6420 CPU class). For example, if the analyst enters

```
Run Prog2
```

Prog2 will run (and migrate within) the 6420 CPU class (that is, between CPUs 1 and 4). This can be overridden if a CPU class is specified in the **Run** or **Batch** commands. For example, if the analyst enters

```
Run Prog2 CPU = 6410
```

Prog2 will then run and migrate within the 6410 CPU class.

For realtime environments, the most common configuration is to reserve one or more CPUs for realtime applications and define the rest of the CPUs as the \$any CPU class. This allows the realtime applications to have guaranteed access to the register sets of the CPU on which they execute, while still providing both normal timesharing access and process migration across the rest of the system. Realtime processes can be locked into a particular CPU with Pro\$LockInCPU. (For more information, see the Help file on this intrinsic.)

However, a CPU that is heavily loaded with nonmigratable realtime processes has a high load factor because the Register Set Manager may have an abbreviated list of processes available for migration. This can cause an imbalance of the CPU workload that the Process Manager cannot adjust. Load-leveling works best in a system in which there is not a significant amount of "explicit residency" (that is, processes explicitly assigned to particular CPUs).

CPU FailSoft Recovery

When a CPU halts or goes off-line, the Service Processor automatically redefines the CPU class that CPU is part of so other processes on the system can still refer to it. The SVP sends a message to the Register Set Manager on one of the functioning CPUs. The message announces that the CPU is not functioning and requests that the Process Manager migrate the processes on the dead CPU to the live CPU. The SVP allows the dead CPU's CPU subclass to remain in place, but rather than having this now defunct CPU subclass point to itself, it is modified so that it points to one of the functioning CPUs.

TIMER SERVICES

The relevant routines for control of the timer services are `$PauseFor`, `$PauseUntil`, `SM$GetTimerLink`, `SM$SetTimer`, and `SM$SetTimerCount`. For more information on each of these, please see the corresponding Help files. It is important to note that `SM$SetTimer`, which does not allow the user to specify a count, specifies a count of 65,535; after 65,535 timer messages are sent, it stops.

For simple operations, use `$PauseFor` and `$PauseUntil`, both of which pause the process for the specified amount of time. These intrinsics rendezvous with the Timer Manager the first time they are called and then maintain the link to the Timer Manager thereafter. They send a timer request each time. There are two steps in performing timer requests.

1. The process must get a link to the Timer Manager. Each link keeps exactly one outstanding timer request going at a time. The process can use the link any number of times, but a process can only have one request per link. (Although a process can get multiple timer links by issuing multiple calls to `SM$GetTimerLink`.) There is usually little reason to have multiple links to the Timer Manager and there is no reason to discard the link until the process is finished.

System Foundation

If you have a program that is going to use timer services (other than calling `$PauseFor` and `$PauseUntil`), start by getting a link with `SM$GetTimerLink` and then hold on to that link for the duration of the process. Also, `SM$GetTimerLink` passes the Timer Manager the link that it uses to send the process timer messages.

2. The process sends a timer request. The timer request can specify these items: 1) the time at which you will get a message back, 2) the number of messages you will get back starting at that time, and 3) an interval between those multiple messages. A process can only have one timer request for any link the process holds.

You can specify exactly one message at a certain interval (for example, 1 timer message in 10 seconds). You can also specify the number of messages (up to 65,535) and an interval between them (for example, 300 messages starting 2 minutes from now with 1 second between each message). You can reprogram a timer at any time. To effectively disable a timer, reset it for a long time in the future. To stop the timer entirely, delete the link to the Timer Manager, which will in turn delete its link back to your process.

If you need wake-ups or messages at short intervals, the most efficient thing to do is to ask for a repeating timer. For example, request 200 messages at 10 millisecond intervals. If your process receives messages very rapidly, it is not a good practice to ask for thousands of timer messages because this may overload the message buffers.

If a program asks for an excessive number of timer messages at rapid intervals, it may not be able to receive them as rapidly as they are sent. Eventually, the Timer Manager will begin to receive errors when it sends the timer messages, but continue to send them. Since these error messages are ignored, the process may lose some of the messages. When this occurs, the Timer Manager is blocked, and the system is gradually brought to a halt. This can happen to any system process; it is just unusual for it occur with a system service other than the Timer Manager.

To avoid having this happen, have your process request a moderate number of timer messages (such as 100) and when this group of 100 messages have been received, send a request to the Timer Manager for another set of 100 timer messages. Thus, if your process is not running or is unable to receive messages, your process will not send the next request, limiting the amount of message buffers in use at any one time.

The only time the Timer Manager may get blocked and be temporarily unable to respond to realtime timer requests is when the system administrator issues a **ReopenProfile** command. At any other time, the Timer Manager interrupts whatever it is doing to respond to timer requests. The Timer Manager is not normally frozen in memory.⁶

Improving the Accuracy of the Interval Timer

The only thing one can do to improve the accuracy in the interval timer service (other than freezing the Timer Manager in memory and locking it in a register set) is to communicate directly with the hardware in the other CPUs. To do that, you must configure your process in such a way that it has a configured link (that is, an ELCONed link) to the hardware on the other CPUs.

The most accurate way to deal with timers is to 1) have the user's process frozen and locked in a register set on a CPU other than the primary CPU, 2) have a To-hardware link directly to the CPU, and 3) send the timer messages directly to the CPU. This procedure does have the complexity of ELCONing the process into the system, locking a register set in that CPU, and arranging the To-hardware link. Once a process has a To-hardware link, it is not restricted to only using the timer services. A To-hardware link to the CPU gives a process extraordinary powers and is therefore an arrangement that should be handled with great care.

⁶ The system can be configured to lock the Timer Manager in a register set. Please note that the Timer Manager is called the "System Profile Manager" in the configuration files.

System Foundation

The ReadCPUTime instruction returns the cumulative CPU time and the ReadRealtime instruction returns the time of day. (The times are in 25 nanosecond intervals.) You can execute these instructions by calling the OS\$ReadCPUTime or OS\$ReadRealtime intrinsics. (For more information, see the Help files on these intrinsics.)

This chapter has described the principal elements required to understand the System Foundation, including the system intrinsics, processes, scheduling and CPU management, and CPU classes. The next chapter discusses the memory system on the System 6400 and how to manage it.

CHAPTER 4

MEMORY MANAGEMENT

INTRODUCTION

The ELXSI System 6400 provides a sophisticated virtual memory system that allows each process on the system to use up to two gigabytes of private memory, plus an additional gigabyte of shared public space memory. The different operating systems use the virtual memory address space in different ways.

The amount of virtual memory is independent of the actual amount of physical high-speed memory hardware installed on the system. Currently the maximum physical memory is limited to two gigabytes, while the total virtual memory is limited only by the amount of disk space available. Special microcode and hardware work in conjunction with the ELXSI System Foundation to transparently move a process's virtual memory between disks and high-speed memory (cache) as needed. Sophisticated memory management algorithms maximize the probability that a given virtual memory location will be found in high-speed memory when it is needed.

The System 6400 provides extensive support for sharing memory between multiple independent processes, a critical need in parallel processing and in many realtime applications as well. The existence of a memory cache on each CPU becomes an important consideration in these applications.

VIRTUAL MEMORY

The instruction set of the System 6400 supports a 32-bit memory address, which provides a virtual address space of four gigabytes. The Memory Manager (and other parts of the System Foundation) shuffle pages between disk and physical memory and map a process's virtual memory to the correct addresses in physical memory.

By convention, the four gigabyte virtual address space is divided into four sections or *subspaces* commonly referred to as P0, P1, P2, and P3, as shown in Figure 4-1. P3 is reserved for future architectural enhancements and is not discussed further here. The P2 subspace (more commonly called *public space*) is shared among all the processes running on a System 6400. That is, any reference to an address in P2 by any process refers to the *same* physical memory location. Public space is normally kept read-only, although it is possible to configure parts of it as writable when the unsecure nature of data in public space is not an issue.

The most common use for public space is for shared libraries. System library code is installed (copied) into public space where it is available to all processes. This saves both memory and disk space, and increases the probability that library pages will be in memory when needed. Embos and EMS also use P2 for shared program code. (ELXSI UNIX systems use a different method for shared programs, although they do use P2 for shared libraries.)

The areas designated as *private space* (P0 and P1) are reserved for the private use of each process. Each process has unique access to these two gigabytes of its address space. Private addresses start at 0 and proceed upward to #7FFFFFFF; most processes, however, use only a small fraction of this address space.

Each process usually has a number of static data structures and some dynamic data structures. Examples of the static data structures are global variables declared at compile-time, constants, debug records, and the code itself. Examples of dynamic data structures are the stack,

Memory Management

which is used for procedure calls and local variable information, and the *heap*, from which space can be dynamically allocated and deallocated.

In Embos, P0 starts with global data and map space, followed by the heap space, which is allocated as an application requires it. All the subsequent space in P0 is left open for use by the heap. Page 0 is reserved in order to assist in debugging programs that mistakenly try to access through an uninitialized (that is, zero) pointer.

At the high end of P1, we find the Info Page, code, constants, and debug records. Directly below that is the stack data area, which like the heap, has room to grow as applications needs it. Typically, there is a large segment of virtual address space between the heap and the stack that is left free to allow those two areas to grow as needed. To avoid “runaway” programs that exceed the stack and heap limits (and possibly use a large amount of CPU time before terminating abnormally), the Binder allows you to place precise limits on the size of heap and stack.¹

When debugging on the System 6400, it is possible to identify the subspace a particular address is in by identifying the upper digit of the address, as shown below.

P0: #0000 0000 to #3FFF FFFF
P1: #4000 0000 to #7FFF FFFF
P2: #8000 0000 to #BFFF FFFF
P3: #C000 0000 to #FFFF FFFF

Virtual memory is further divided into pages of 2048 bytes each. Each page of virtual memory can be located either in physical memory or on

¹ For more information on stack and heap limits, refer to the Help file for the Bind command.

Memory Management

the disk, and has access rights (such as read, write, and execute) associated with it. The Memory Manager (MM) determines which pages can be in physical memory and which should be flushed out to disk. It also provides services that allow a process to control its own pages.

VIRTUAL MEMORY UNDER UNIX

Virtual memory under UNIX is similar to memory under Embos, but there are also significant differences. For example, P0 and P1 are organized somewhat differently: the code and constants pages reside in P0, but the BSS (which is analogous to the Embos Uninitialized Variables section) grows upward in P0 and the expandable Stack grows downward in P1 just as they do in Embos. For a diagram of memory under UNIX, see Figure 4-2.

P0

P0 is divided into the following areas:

1. The first page (Page 0) is reserved (no access).
2. The Text area, which is all code (read/execute).
3. The Data area, which is initialized data (read/write).
4. The BSS area, which is uninitialized allocatable memory (read/write).

Information usually grows from the high end down in P1 (that is, from the largest address to the smallest address). The stack starts at some predefined location in P1 and grows toward the BSS. The heap under UNIX begins immediately after BSS. Memory can be allocated from the BSS with one of two system calls: **BRK** and **SBRK**. They essentially perform the same task, which is to move the end of allocated memory up or down.

P1

The first location used for the stack is called the Stack Base. It is preceded by the area called P2_Globals, in which a number of local variables reside: Errno, CPUBaseTime, H_Errno, Environ, _IOB, _SmallBuff,

`_SIBuff`, and `_SOBuff`, and so on. These variables are the globals and statics for the UNIX public space libraries. This region is equivalent to the System Static Data area for Embos processes.

`Errno` is the global error number that is returned at the end of every I/O operation. The `CPUBaseTime` is updated for each new invocation of a process skeleton. `H_Errno` (BSD only) is the global error number that is returned at the end of every networking operation. `Environ` is the pointer to the user's environment. `_IOB` is the base array I/O buffer structures (not the buffers themselves). `_SmallBuff` is used with the BSD I/O library to keep a one-character buffer associated with each one of the allocated `_IOBs`. `_SIBuff` and `_SOBuff` are full-sized buffers.

The Lifeline Handler Base is the base of the Stack that the Lifeline Handler switches to when it receives certain types of messages from the kernel. Instead of working on the user Stack, the Lifeline Handler Base has its own one-page stack. The Arg-Cache is an area in which the arguments to a command are copied before an `Exec` system call is issued.

P2

P2 has two parts: an EMBOS region and a UNIX region. The UNIX region is further divided into six parts, two for System V, two for UNIX 4BSD, and two System Foundation Branch Tables. The separate System V and 4BSD regions provide four nonconflicting public libraries, all of which are accessible at the same time.

The UNIX section keeps the public libraries (*libc.a*) and public programs that help to run the UNIX system. The System Foundation Branch Tables provide non-relocating entry points to the System Foundation intrinsics that are called for sharing memory, realtime tasks, and so on. The purpose of the Branch table is to eliminate the need to relink ENIX processes when a new Embos/System Foundation public space is installed with each new Embos software release. The Branch tables are also used for accessing UNIX shared libraries.

Memory Management

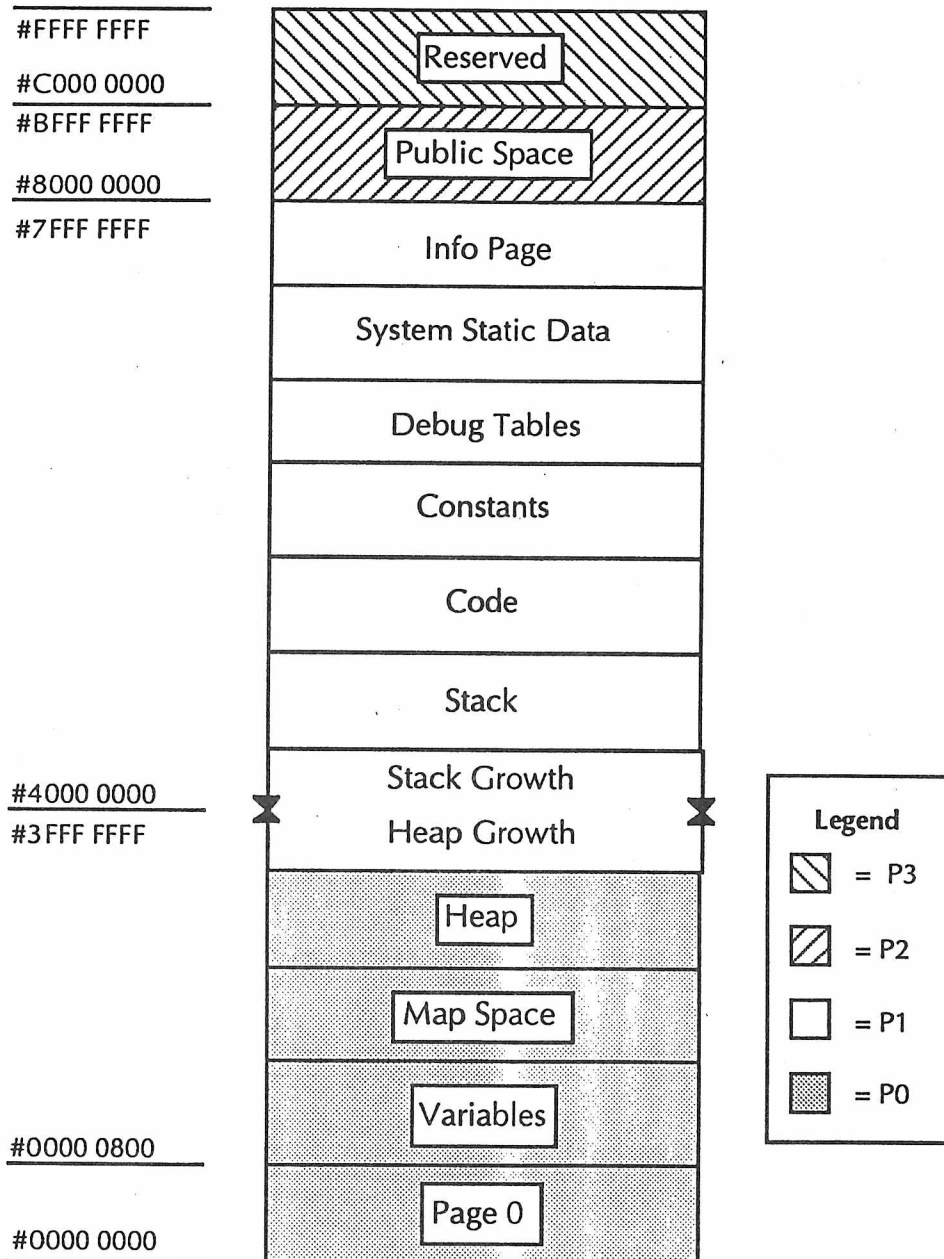


Figure 4-1. The Virtual Address Space Under Embos and EMS

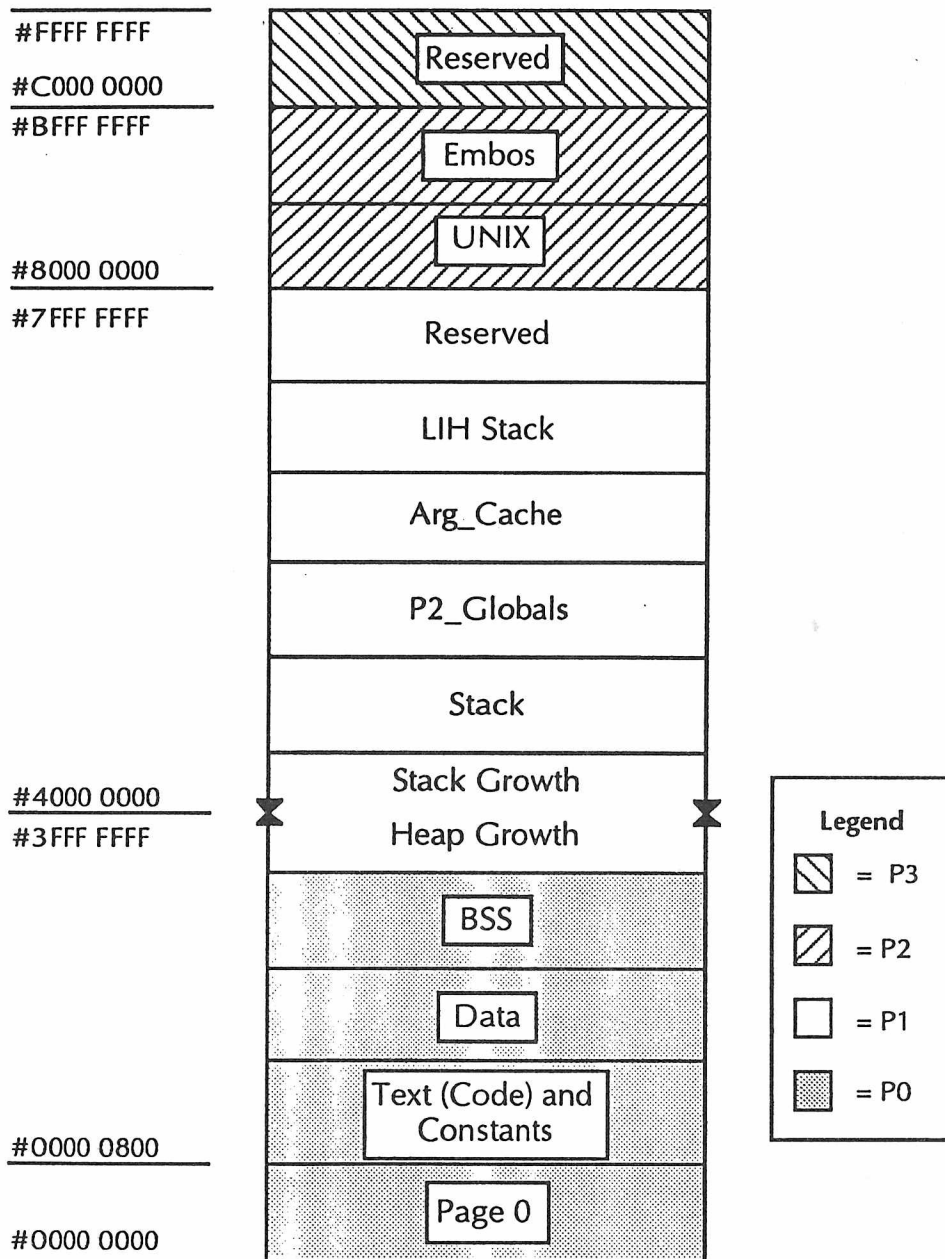


Figure 4-2. The Virtual Address Space Under UNIX

PAGE MAPS

A page map is a data structure used by the hardware that defines where each of a process's pages of virtual memory is located, either in physical memory or on the disk. Each page map entry describes one page of virtual memory, including its location and access rights. For each subspace in virtual memory, there is a separate page map.

If a page map were a simple array of page map entries, supporting a virtual address space of a gigabyte would require a page map several Megabytes in length. To minimize the amount of physical memory required for the page map data structure, it is divided into a hierarchical structure of multilevel page maps. The advantage of multi-level page maps is that page map entries for unused pages can be omitted, and page map entries for infrequently-used pages can be migrated to disk when not needed to save physical memory. For example, page map entries for the unused address space between stack and heap can be omitted. A multi-level page map data structure can cover a wide range of virtual addresses and still only allocate table space for the addresses being used.

Each page map page has 256 entries, each of which maps one 2-Kbyte virtual page to a physical page. A *level-1 page map* points directly to memory. A single level-1 page map can describe 512 Kbytes of virtual memory.

A *level-2 page map*, instead of pointing to virtual memory locations, points to level-1 page maps. A level-2 page map can point to 256 level-1 page maps, so the total space that can be described by a level-2 page map is 128 Mbytes.

For some processes (such as a process that maps a file larger than 128 Mbytes), a level-2 page map may not be large enough to describe the virtual memory required for a single process. In such cases, a *level-3 page map* is required. A level-3 page map consists of eight pointers to level-2 page maps, which in turn point to level-1 page maps. The total amount of virtual address space that can be addressed by a level-3 page map is $8 \times 256 \times 512 \text{ Kbytes} = 1 \text{ gigabyte}$.

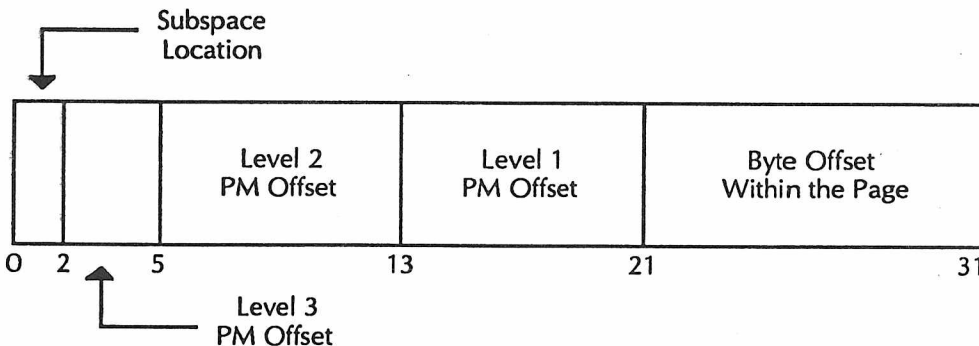
When a process is spawned, the Memory Manager builds the page tables for level-1, 2, or 3 page maps as required by the size of the process. Under Embos, this information is kept in the InfoPage.

THE VIRTUAL ADDRESS

The 32-bit virtual address is divided into several different segments. The least significant eleven bits is the byte offset within a page. The two most significant bits indicate whether the page is in public or private space:

00 = P0, 01 = P1, 10 = P2, and 11 = P3

Each subspace has its own page map table. The page map structure within each subspace is a hierarchy that starts with one root page map and may extend to multiple lower-level page maps. Whenever there are multiple level-n page maps in a subspace, a level n+1 page map is required to point to the level-n page maps. The CPU uses these upper two bits in the virtual address to determine which subspace page map table to start with.



A page map is one page (2 Kbytes, 256 words). Each word (64 bits) in a page map contains a page map entry. Each page map entry in a level-1 page map points to (that is, holds the physical address of) a target page, which is the actual page in virtual memory. Thus, a single level-1 page

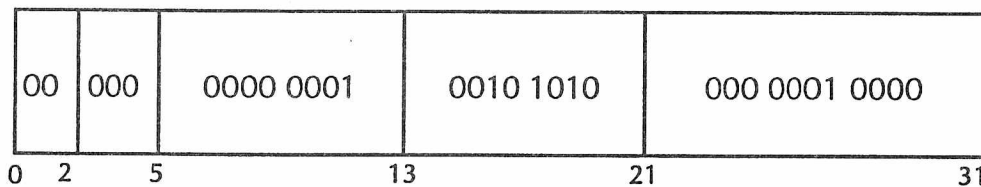
Memory Management

map can address at most 256 target pages (512 Kbytes or .5 Mbyte). The Process Control Block (PCB) contains the physical address of the *root page map*. Each page map entry in a level-2 page map points to a level-1 page map; and each page map entry in a level-3 page map points to a level-2 page map.

A software process can have a single level-1 page map as its root page map for subspace P0. This root page map can address a range of addresses from #0000 to #7 FFFF. You can produce an application that uses no more than 512 Kbytes of virtual memory by limiting the number of global variables and specifically defining the number of Map Space pages and the maximum number of heap pages. If the total amount of space for P0 does not exceed the 512 Kbyte boundary, the application operates in a one-level P0 subspace.

If an application requires a P0 larger than 512 Kbytes, additional level-1 page maps are necessary to access this larger address space. Suppose the maximum P0 address that the program will use is #7FF FFFF. This requires 256 level-1 page maps and one level-2 page map. This now becomes a two-level subspace, and the PCB identifies the root page as this level-2 page map. The entire P0 subspace requires eight level-2 page maps; thus, an application that needs the entire range of addresses in P0 requires a level-3 page map as the root page.

Let us take as an example a **Load** instruction that accesses the following virtual address: #0009 5010. Breaking this address down into its constituent bits, the address looks like this:

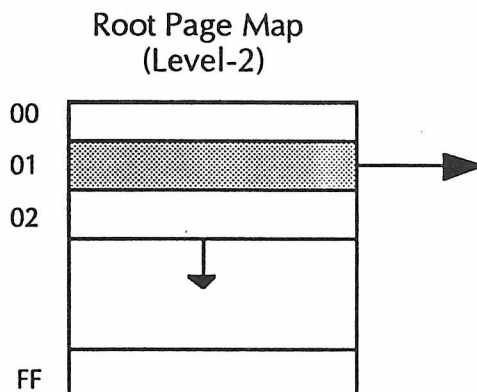


Memory Management

Analyzing this address further, you can see that it gives the following information:

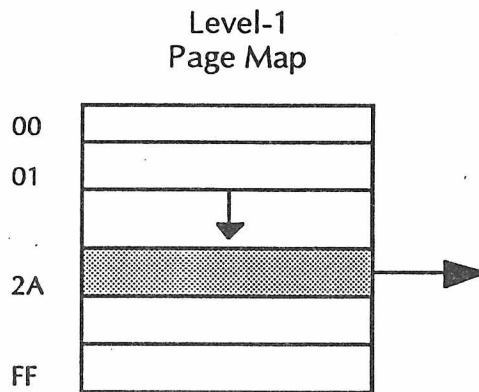
- Subspace location is 00, which indicates subspace P0.
- Level-3 offset (next three bits) is 000.
- Level-2 offset (next eight bits) is 0000 0001.
- Level-1 offset (next eight bits) is 0010 1010 (#2A).
- Byte offset within the page (last eleven bits) is 000 0001 0000 (#10).

Because the level-2 offset is non-zero, the subspace must either be a two-level or a three-level subspace. If the Process Control Block indicates that this is a two-level subspace, the root page is a level-2 page map, as shown in the illustration below.

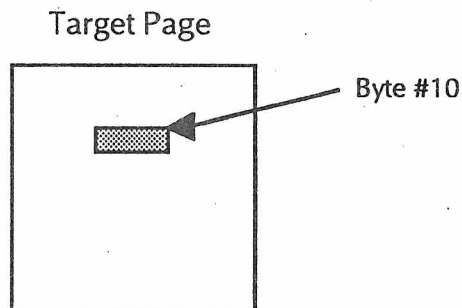


Memory Management

The level-2 offset finds the page map entry that identifies the level-1 page map.



The level-1 offset within that page map finds the page map entry that points to the target page. Then the byte offset within the target page accesses the byte to be loaded.



If using this same address, the Process Control Block indicated a level-3 subspace, a level-3 offset of zero would find the page map entry that identifies the level-2 page map.

PAGE FAULTS

The Memory Manager reads the page map entry to find out where a physical page is. To the microcode, there are only two possible states of a virtual page: in or out of memory. Those states are toggled by the most significant bit (Bit 0) of the page map entry. If Bit 0 has a value of 1, the page is in memory and any byte in that virtual page can be accessed. If Bit 0 has a value of 0, the page is out of memory and any attempt to access that page causes a page fault.

When a page fault occurs, the microcode sends a message to the Memory Manager that includes the process ID of the faulting process and the virtual address that the process attempted to access. The Memory Manager determines what type of virtual page it is and how to bring it into memory. Some pages reside on disk, so the Memory Manager must read the page in from disk; other pages, such as new heap or stack pages, are simply given a page in physical memory. The latter type of page fault is resolved much faster than one that requires reading pages from disk.

Once the page fault is resolved, the Memory Manager indicates that the virtual page is "in-memory" and instructs the microcode to continue execution. The process reexecutes the instruction that caused the page fault (which is typically an operation that loads or stores into memory) and resumes where it left off.

Memory Management

SHADOW FILES

Shadow files are files on disk to which pages of virtual memory can be written when (or if) the Memory Manager takes these pages out of memory. Each process has a shadow file.² Shadow files are the only residence on disk for all the forms of dynamically allocated memory. Shadow file space is allocated on a first-come, first-serve basis. Each page of memory that needs to be written to disk is assigned the next available page in the shadow file.

In theory, every virtual page that is mapped to a physical page and has been modified has a disk address that it can be written to. In practice, however, the Memory Manager delays assigning a disk address to a page in memory as long as possible, and only does so when a write to disk is imminent. This avoids allocating unnecessary shadow files, since many pages are never written out to disk.

Under Embos, an executing program receives most of its pages (code, constants, initialized variables, and so on) from the Bound or executable file. When the program page faults on one of those virtual pages, the page in question is read from the Bound file. Most pages that are read from the Bound file do not need to be written back to disk because they usually are not modified. But if data is modified, it cannot be written back to the Bound file (because to do so would destroy the initial values of the file). Instead, such a page is written to the process's shadow file. One case in which a Bound file page is modified is when a Debug breakpoint is set in it; another case is when an initialized global variable is modified.

Under UNIX, there is only a single Bound file that is shared among all the user processes. This Bound file is a program called *Init*; it is found in the controlling directory of each UNIX system. This program is executed during a UNIX boot, and it immediately Executes the UNIX Init

2 All the processes configured by Elcon share one shadow file called the System Shadow File.

program. All other processes are created by forking Init, and subsequently by forking children of Init. A fork creates a new process (and shadow file) but retains the underlying Bound file. To execute programs other than Init, use the `exec` system call, which writes a new executable image on an existing process.

The Text section is normally shared among all the concurrent user processes, and consequently, it does not occupy space in the shadow file unless a text page is written to (for example, to insert a Breakpoint instruction). The Data section is copied from the file being Exec'ed to the portion of the process's address space beginning at the first page boundary that follows the Text section. The page maps for the rest of the process's address space are adjusted so that they cause a nondisk page fault when first accessed, which is satisfied with a "virgin" memory page.

A unique aspect of the System 6400 is that paging can occur against user-accessible files as well as against shadow files. This mechanism is called "mapped files" and is discussed in detail at the end of this chapter.

KEEPING DATA IN MEMORY

There are typically more virtual pages that have been accessed than there are physical pages. The Memory Manager's job is to equitably assign physical memory pages to virtual memory pages, minimize page faults, maximize throughput, and attempt to predict which virtual pages will be needed in memory.

Developers of realtime applications require guaranteed access to memory and cannot afford to have page faults. To make this possible, the Memory Manager supports services to "freeze a process" and "freeze pages." When a process informs the Memory Manager that it is a frozen process, every virtual page this process accesses is retained in memory until the process terminates (or is unfrozen). Alternatively, specific pages can be frozen in memory.

Memory Management

Let us suppose a process starts to execute and requests to be frozen via the MM\$FreezeProcess intrinsic.³ If the process has sufficient security access, the process will continue to execute. The first time the process requires a page that is not in memory, the Memory Manager brings that page in and, if that page is in private space, freezes it in memory. If a frozen process accesses public space, the public space virtual page will be brought into memory, but that physical page will not be frozen into memory.

For some realtime applications, encountering even initial page faults may not be acceptable. There is another intrinsic to handle this requirement called MM\$FreezeandTouchProcess,⁴ which freezes all the pages in the process's Bound file.

CACHE CONSIDERATIONS

Cache is a small, very fast form of memory that is placed between a CPU and main memory. Cache memory significantly reduces the time spent waiting on memory accesses. Blocks of recently referenced data are placed in the cache with the assumption that it is likely that subsequent references will be to data nearby. When cache satisfies references to memory, the overhead of accessing main memory is eliminated. This frees the Gigabus for DMA or activity across multiple CPUs and significantly improves the cost/performance ratio of the memory hierarchy.

Each CPU in the System 6400 has a high-speed cache memory. On a 6410 CPU, there is a total of 16 Kbytes of cache; on a 6420 CPU, there is 64 Kbytes of cache. Cache memory relates to main memory just as

3 MM\$FreezeProcess freezes a process's entire address space into physical memory. For a full description of this intrinsic, refer to the Help file.

4 MM\$FreezeAndTouchProcess freezes a process's entire address space into main memory and then, in order to eliminate subsequent page faults, touches most of the address space. This intrinsic does not bring in additional heap space, so the application must explicitly page fault or touch a block of heap space to bring it into memory. For a full description of this intrinsic, refer to the Help file.

Memory Management

main memory relates to disk storage. The data in the cache is a subset of the data in main memory. Just as data on main memory is backed up to disk files (and managed by the Memory Manager), so is data in the cache backed up to main memory (and managed by the CPU hardware and firmware).

Accessing main memory is relatively slow for two reasons: First, the memory chips themselves are slower than cache memory chips. Second, accessing main memory requires interaction across the Gigabus and needs several Gigabus cycles to complete the operation. Access to the CPU's cache can be accomplished in a much shorter time.

The CPU allocates cache memory and decides when the data in the cache must be written back to main memory. When a process is not sharing memory, it is safe for it to update the copy of its data in the cache and not instantly update the copy of the data in main memory. Problems associated with cache memory only occur when memory is being shared among multiple processes.

Suppose, for example, that two processes executing on two different CPUs are both accessing the same area of memory. Each process must access the memory location through the cache of its CPU. If Process A changes a memory location, that change must be made not only in the cache of Process A's CPU, but also in main memory and in the cache of Process B's CPU. Before Process B can read the new value, Process A must first write the data in its cache to main memory (an operation known as a *cache flush*). Also, Process B must invalidate its cached version of the data to ensure that when it reads the virtual location it will do so from main memory, instead of using the old (and incorrect) value already in the cache. Otherwise Process B will not see the new value of the memory location. When data is changed in a CPU's cache but not in memory (or changed in main memory but not in a CPU's cache), the obsolete value is referred to as *stale data*.

Memory Management

TRADITIONAL CACHE SCHEMES

Several schemes have been implemented on more traditional systems to circumvent the problem of stale data. One scheme is called a write-through cache. This means that every Write to cache memory is automatically written to main memory as well. This scheme also requires that all the CPUs must watch the system bus for write-through operations so that if the location that is being changed exists in their own cache, they can also make a copy of the updated data.

In this scenario, when Process A changes a location in its cache, the cache automatically initiates a Write to memory. The cache in the other CPU notices this Write to memory and updates its own cache. The problems with this mechanism are a) it requires traffic through the bus every time cache is modified, and b) each CPU requires additional logic to watch the bus to see if a location in cache is being changed. This is sometimes referred to as a bus-following cache.

Even with a bus-following cache, a process may retain data values in its general registers for some period of time before explicitly storing the value to a cache location. Thus, a value can be modified in the register while the other sharing processes remain uninformed.

THE CACHE ON THE SYSTEM 6400

Because most systems make extensive use of shared memory for system tables, the extra cost in the hardware to implement a bus-following write-through cache is justified. However, since shared memory is not the default condition on the System 6400, these solutions are not built into the system. When two processes are sharing the same memory on the System 6400, either the cache is disabled for access to memory pages that are shared (that is, all data access to those pages for both Read and Write operations goes directly to main memory) or else the two processes must flush their caches at points where another process might need the current values.

Memory Management

The advantage of having a cache for data is reduced on the System 6400 when two processes on two CPUs are randomly sharing memory. Since the operating system does not use shared memory and applications usually share only a small portion of their memories, this is rarely a problem.

There are circumstances in which multiple processes can share memory without stale data being a problem. For example, multiple processes may be reading the same area of memory, but not updating it. In this case, it is acceptable to have multiple copies of the data in different caches. Since the data is not changed, the caches do not have to be updated.

For shared-memory multiprocess applications on the System 6400, access to shared memory is either noncacheable or explicitly synchronized. (See Chapter 7, "Parallel Processing.") There are parallel processing routines that allow applications to force main memory to be updated at the appropriate times and force the process observing the changes to memory to reload its cache from main memory. This allows the application to use the cache most of the time and only pay the penalty of going to main memory at explicit points of synchronization.

Cachable and Noncacheable Data

There are two states for every virtual page: cachable (the normal state) and noncacheable. (There are relatively few noncached data structures in the System Foundation.) When a process is working with a page that is in the cache, reading and writing to virtual memory is actually reading and writing to cache. If a particular virtual location is not found in the cache, the process is suspended briefly while the CPU's cache goes to main memory and brings a new cache block⁵ into the cache.

A noncacheable page bypasses the cache. An access to a noncacheable page causes a read or write operation to main memory. These operations are substantially slower than accesses to cache. For example, a

5 Cache blocks are 32 bytes (four words) long.

Memory Management

cacheable read operation such as loading a cached word takes approximately 250 nanoseconds. The same access to a noncacheable page is five to seven times slower.

Why then would you ever want to have a noncacheable page? Typically, you might use noncacheable pages when you want to share data between processes and you do not want to flush the cache explicitly. If two processes share the same physical pages, and those pages are cacheable, each process may update locations that are in their respective caches, but main memory may not be updated to reflect the changes. Also, the System Profile keeps its tables noncacheable so as to not disrupt the user process's cache. Noncacheable access does not affect the cache in any way. When you use the **Exchange** instructions to synchronize processes, it is essential that the memory they operate on is shared and noncacheable.

SHARING MEMORY

The System 6400 architecture provides for two types of virtual address space: public and private. Although public locations are addressable by every process, the accessibility of a public page is constrained by the access rights of the page and whether or not it is cacheable. The access rights and cacheability of a public space page is the same for all processes. In general, private memory is not shared between processes. Thus, a process is assured that its address space is protected from other user processes.

When sharing memory, different processes can have pages of virtual memory that point to the same pages of physical memory. One way that multiple processes can share memory is through public space (because public pages are implicitly shared among all processes). Another way to share memory is through memory management intrinsics or system calls that allow a process to define a range of its own address space to be shared with other processes.

Setting up shared memory is analogous to the way that processes communicate via the message system. When a process communicates with

another process through the message system, the communicating process owns the right (called a *link*) to communicate with itself. A process can give a link to another process so that process can send a message back to it. But an outside process cannot arbitrarily access another process through the message system, nor can a process arbitrarily give another process a link into its funnels (that is, the other process must explicitly accept the link by doing a `ReceiveLinkOnChannels` or `ReceiveLink`).

Sharing memory is much the same. A process defines an area of its address space that it can share with other processes. The process can then send a description of these shared pages to the other processes that are expecting to share this memory. (Memory is shared on a page-by-page basis.) The Memory Manager sets up the appropriate page map entries so that the cooperating processes point to the same locations in physical memory.

You can share memory with cached pages, but it requires explicit control by the application to know when to use the `Flush` intrinsics to write data back or invalidate cache areas so that subsequent accesses cause reads to physical memory.

The System 6400 has three forms of shared memory: public-shared memory, private-shared memory, and fork-shared memory.

PUBLIC-SHARED MEMORY

While all public space locations are shared, there are 64 Mbytes of public-shared memory set aside for applications to use. These public-shared memory locations (currently #84000000 to #87FFFFFF) allow Read-Write-Noncacheable access. (The symbol `OS$PUBLIC_DATA` holds the address of this public data area.) Thus, this entire 64 Mbytes of space is implicitly shared memory among all processes.

One advantage of public-shared memory is that it is easy to use. No run-time set up is required and the Read-Write locations are available by simply accessing them. Typically, a FORTRAN programmer creates an object file that contains a symbol mapped to a public address in the

Memory Management

64 Mbyte range. Each program that needs to share memory declares a *labeled common*. To effectively map the labeled common to public space, each program is bound or linked with this special object file.

One disadvantage of public-shared memory is that data is not protected. Any process in the system can modify or otherwise access (either deliberately or accidentally) any public shared location. A further disadvantage is that all access to public-shared memory is non-cachable.

PRIVATE-SHARED MEMORY

Private-shared memory is the most general form of shared memory. The private-shared memory facility permits an application to share a portion of its address space with other processes and to limit the access rights of those cooperating processes to a subset of the owner's rights. There are two types of memory-sharing intrinsics: the `$ShareMemory` intrinsic and the `MM$` intrinsics. In addition, UNIX systems provide system calls that give some shared memory capabilities. These are discussed in the appropriate UNIX documentation. The intrinsics discussed here are System Foundation services and they are accessible to all System 6400 processes, including UNIX processes.

The `$ShareMemory` Intrinsic

The `$ShareMemory` intrinsic allows a process to designate part of its virtual address space as a named shared region. The process does not have to know about any other process or communicate with any other process. A process that uses this intrinsic simply shares a region of memory and gives this region a name. Other processes reference the memory region via the name. The changes to the data in the named region of memory made by one process are visible to the other processes sharing memory (and vice versa). This means that all accesses to the shared region are noncachable.

A region name assigned with the `$ShareMemory` intrinsic is global, but it is only usable by processes that have the same user ID as the creator of the named region. (A process that calls `$ShareMemory` with a new

Memory Management

region name is the creator of that region.) The named region of memory exists until all the process using it have terminated or released it with the `$UnshareMemory` intrinsic, at which time the name is deleted. For more information, refer to the Help file.

The MM\$ Intrinsics

The MM\$ intrinsics provide more control than the `$ShareMemory` intrinsic, but they also require more careful coordination with the sharing processes.

A process called the "owner" declares an arbitrary range of addresses in its virtual address space to be sharable, and explicitly allows specific other autonomous processes, called "nonowner" processes, to obtain access to those pages via a three-way communication between the owner process, the nonowner processes, and the Memory Manager. The nonowner processes need not have those shared pages installed in the same virtual address as the owner process pages.

This presumes two things: 1) the processes must coordinate their activity to the extent that the owner has a message system link to each nonowner(s) and 2) to permit the exchange of data, the owner and each nonowner must program a mutually-defined message protocol between them.

The owner process uses the `MM$OfferSharedPages` intrinsic to define those sharable pages and then sends a message to a nonowner process to inform it that the shared memory is available. Finally, the nonowner process uses the `MM$BidForSharedPages` intrinsic to map the shared memory into a portion of its own address space.

The owner process uses the `MM$OfferSharedPages` intrinsic to specify the following parameters:

- *SharingBaseAddress*. A page-aligned private virtual address.
- *SharingByteCount*. A number of bytes that is a multiple of the page size (2,048 bytes).
- *AllowWriteAccess*. If this is set to True, the subsequent sharing processes are given write access to the shared memory.

Memory Management

- *AllowCachedAccess*. If this is set to True, the subsequent sharing processes are given cachable access to the shared memory.

The *MM\$OfferSharedPages* intrinsic returns a 16-bit Link ID called the *ShareLink* that is associated with this sharable region and its attributes.

The owner uses the *MS\$CopyLink* intrinsic to copy the *ShareLink* to a nonowner. The nonowner process in turn receives the *ShareLink* using the *MS\$ReceiveLink* (or *MS\$ReceiveLinkOnChannels*) intrinsic. (For ways to access these instructions from UNIX programs, see *man e_ops*.) The nonowner process then uses the *MM\$BidForSharedPages* intrinsic to specify the following parameters:

- *SharingBaseAddress*. A page-aligned private virtual address in the nonowner's address space.
- *SharingByteCount*. A multiple page number of bytes that must not exceed the size of the owner process's shared region of memory.
- *ShareLink*. This identifies the set up of the owner's shared pages. It must be the link passed from the owner for this purpose.

FORK-SHARED MEMORY

A *process fork* is a system service that produces a new process by effectively duplicating a process's address space (that is, its code and *un-shared* data). The child process receives a "snapshot" copy of the parent's address space, duplicating its access rights, cachability, and current page contents. The virtual address space that is shared among the forked processes maps to the same physical address.

Fork-shared Memory is closely tied to the parallel processing services provided on the System. If a parent process declares a range of address space to be sharable and then forks, both the parent process and child process have shared access to that specific range of shared pages. Subsequent forked processes will also have access to these shared pages. Pages that are not shared are replicated in the child's address

Memory Management

space, but the parent and child processes have different physical pages and can therefore modify their data without affecting the other process.

The parent process can use the `MM$SharePages` intrinsic to specify that page-sized fragments of its address space be marked as “shared”; a subsequent fork gives the child process shared access to those pages. This technique is used by the `MT$ (ShareMemory)` parallel processing intrinsics.

Fork-shared memory is useful, if not mandatory, for parallel processing applications. But the scheme is limited to applications that fork to create multiple, cooperating copies of a single program.

To synchronize processes using shared memory under parallel processing, there are two things to bear in mind. First, if you share memory, you can designate it as cachable data or noncachable data. If the data is not going to be modified by the sharing processes (which is the simplest case), there is no need to synchronize access to the data.

Second, if the data is noncachable and is going to be modified at any point, you must synchronize the access to that data. When you modify a location, the compiler generates a Load from memory into a CPU General Register, the operation itself, and then the Store back into memory. Although this is only one line of source code, it is often at least three internal instructions, which can therefore be interrupted. The Exchange instructions are the only uninterruptible memory instructions that read and modify a location in one atomic operation.

Memory Management

SHARING MEMORY UNDER UNIX

Sharing memory under UNIX is accomplished in the same way that it is under Embos--by calling the same Embos intrinsics from UNIX. Many of the Embos public space intrinsics can be called directly from UNIX programs. However, before a UNIX process can call Embos public space intrinsics, some initialization must take place within the UNIX process. First, the UNIX process's address space, which differs significantly from that of an Embos process, must be adapted to look like an Embos process's address space. For example, if you are going to allocate Heap via the Embos HM\$ routines, locations in the process's System Static area must be initialized correctly.

Second, if the Embos intrinsic you are calling from a UNIX process needs to operate in the Embos NameSpace, the UNIX process needs a link to the NameSpace Manager (NSM). UNIX processes do not normally have this link. Two questions arise here: How does a UNIX process get an NSM link and what Embos user ID will be associated with that link (and thereby that process)? The ELXSI-Link Driver (E-L Driver) allows a UNIX process to get a link to the NSM.

A driver in UNIX is part of the kernel that interfaces with a device or provides a device-related service. In this case, the service the E-L Driver provides is to give a UNIX process access to the NameSpace Manager.

6

When a process requests the NSM link from the E-L Driver, it must, as part of the request, specify the Embos user ID and password to be associated with the link. The E-L Driver verifies the user ID and password. If they are legal, the driver informs the NSM to change the Embos user ID associated with the UNIX process to the specified Embos user ID. The driver then provides the UNIX process with an NSM link. Now the

-
- 6 The E-L Driver also allows two UNIX processes to get message system links to each other in a manner similar to the simple rendezvous service provided by the NSM from Embos. See the UNIX manual page for the *el driver* for a complete description of this driver's functions.

UNIX process can open Embos NameSpace entries such as rendezvous, disk files, device server nodes, and so on.

The UNIX library *libel.a* contains several functions to aid the UNIX process in initializing the UNIX process's address space and obtaining the NSM link. A call to the function *emboslogin* performs both these tasks.⁷ Once this is done, the UNIX process can access Embos intrinsics, including memory-sharing intrinsics.

By default, *emboslogin* allocates 50 Kbytes each of both heap and map space for use by the Embos intrinsics. For some programs, this may be inadequate. In such cases, your application should call the function *embosinit* before calling *emboslogin*. *Embosinit* only initializes the address space and allows you to specify the amount of heap and map space. You can specify less than 50 Kbytes for each if desired. To obtain the NSM link, *emboslogin* must still be called after *embosinit*. *Emboslogin* notices that the address space has already been initialized and does not perform it again.

Because the intrinsics in the Embos public space are written in Pascal and conform to the Pascal parameter-passing conventions, a process cannot directly call these intrinsics from C in UNIX.⁸ To make the necessary parameter conversions, the system provides a special interface function called *elxsi_call*.⁹

There are currently two types of Embos intrinsics that a UNIX process cannot call: 1) intrinsics that perform I/O to the user's terminal, including any of the standard input or output descriptors when these descriptors are associated with the process's terminal, and 2) intrinsics that spawn or fork a process in Embos.

7 For a complete description of this and other functions provided within *libel.a*, see the UNIX manual page for *emboslogin* (3).

8 In general, Embos Pascal passes its parameters in registers and UNIX C passes its parameters on the stack.

9 For a complete description of this function, see the UNIX manual page for *elxsi_call* (3).

Memory Management

DATAPOOLS

A *datapool* is a collection of variables that are shared among cooperating processes and exist in the private address space of each of the cooperating processes. But unlike data stored in disk files, datapool variables exist only for the life of the job; once the last process using the datapool terminates, the values of datapool variables are lost. Each datapool contains a symbol table, called the *datapool descriptor*, from which one can find out the name, type, and location of all the variables in the datapool. Lastly, variables can be moved around within the datapool, added to, or deleted from the datapool without having to recompile those programs that use the datapool.

The datapool product has four parts:

- *Datapool Assembler*. Builds variable layout information.
- *Memory sharing*. Shares variables between processes.
- *Datapool intrinsics*. Return information on datapool.
- *Language interface*. Provides compiler support.

DATAPOOL ASSEMBLER

A complete description of a datapool is placed in a *.dp* file. The Datapool Assembler (DPasm) processes this file and generates two elements: a set of *binder directives* for linking and storage allocation and a *symbol table*, which holds the names, addresses, offsets, types, and sizes of all the datapool variables. All of this information is placed in an object file. This object file must be bound in with all programs that use variables in the datapool.

The Binder returns the address of the symbol table in the variable

`<DatapoolName>_DPDESC`

This datapool descriptor variable is very much like a file descriptor in that it points at a control block and must be passed to the DP\$ intrinsics.

The DATAPOOL statement in FORTRAN automatically declares this variable. Pascal and C have no such feature and therefore the program-

Memory Management

mer must explicitly declare the variable. (Note that both the table and the pointer to the table are called datapool descriptors; it will always be clear from the context which is meant.)

For more information on the Datapool Assembler and the syntax of the the *.dp* file, type

Help DPasm

DATAPOL INTRINSICS

The datapool intrinsics (the names of which begin with DP\$) return information about a datapool and its symbols. Typically, the datapool descriptor generated by DPasm must be passed to these intrinsics.

The two routines you will certainly need are :

```
FUNCTION DP$DataPoolAddress
  ( VAR DPD : DP$DescType ) : integer;
{ Returns the address of the datapool. }

FUNCTION DP$DataPoolSize
  ( VAR DPD : DP$DescType ) : integer;
{ Returns the size of the datapool. }
```

These two intrinsics return the address and size of the datapool. This information is passed to \$ShareMemory in order to make the datapool sharable.

Memory Management

LANGUAGE INTERFACE (PASCAL, FORTRAN, C)

Datapool variables should be declared in such a manner that their names are exported to the Binder: in C and PASCAL, declare them as global external variables; in FORTRAN, declare them with the DATAPOOL statement. To avoid name conflicts between datapool variables and other program variables in C and Pascal, we recommend that you prefix the datapool name to all the variable names.

For example, if *up* is a variable in the datapool called *sim*, we recommend you declare it as *sim_up*; the Datapool Assembler (DPasm) produces the names this way by default. (See "Name Scoping Datapool Variables" and the specific language examples below.)

Programming Examples

In Pascal, one need only declare datapool variables as global external variables. For example:

```
%include /embos/include/DP$Routines.p
{ Datapool variables }
{$E+ -- Make the variable names externally known.}
{$V+ -- Make the variables "Volatile," that is, }
{      memory resident. }
VAR
  sim_dpdesc      : DP$DescType;
  sim_dir         : char;                { direction }
  sim_up          : boolean;             { up-down }
  sim_index       : array[1..32] of integer;
                        {gather-scatter index }
  sim_sortlist    : array[1..2] of array[1..32] of char;
                        { names }
  tag             : integer;             { tag from $ShareMemory}
{$E-}             { share it }
tag := $ShareMemory ( 'MySectionName'
                      , DP$DatapoolAddress(sim_dpdesc)
                      , DP$DatapoolSize(sim_dpdesc)
                      );
```

Memory Management

In C, as in Pascal, one need only declare datapool variables as global external variables. However, case-sensitivity is an issue. You will probably want to use uppercase variable names in your C program for all datapool variables. This is because Embos compilers and assemblers (other than C) make the names of external symbols uppercase. (You may instruct DPasm not to make the names uppercase, but then it is not usable by non-C programs.)

```
#include <estring.h>
typedef char[32] aName
int      *SIM_DPDESC;
char     SIM_DIR
char     SIM_UP
int      SIM_INDEX[32]
aName    SIM_SORTLIST[2]
int      tag
estring  sectionname
sectionname = CtoESring ( 'MySectionName' );
tag = $ShareMemory ( @sectionname

DP$DatapoolAddress (@SIM_DPDESC)

DP$DatapoolSize (@SIM_DPDESC)
);
```

Memory Management

In FORTRAN, you declare a common block as a datapool; the compiler takes care of the rest. In particular, the user program does not have to declare the datapool descriptor, *sim_dpdesc*, as shown below.

```
DATAPOOL /sim/dir,up,index,sortlist
INTEGER*1      dir
LOGICAL*1      up
INTEGER*4      index(32)
CHARACTER*32    sortlist(2)
INTEGER*4      tag
EXTERNAL       $ShareMemory, DP$DatapoolAddress,
               DP$DatapoolSize
INTEGER*4      $ShareMemory, DP$DatapoolAddress,
               DP$DatapoolSize
tag = $ShareMemory ( %string 'MySectionName'
+                   , %val
DP$DatapoolAddress(sim_dpdesc)
+                   , %val DP$DatapoolSize
(sim_dpdesc)
+                   )
```

SHARING DATAPOOLS

\$ShareMemory and \$UnshareMemory manage the sharing of the datapool and must be called programmatically. The \$ShareMemory intrinsic associates a user-supplied name to the datapool so that all the processes that want to share the same datapool need only supply the same name. For example,

```
$ShareMemory ( 'flight_sim', DatapoolAddress,
DatapoolSize )
```

associates the section name *flight_sim* with the specified region of your private address space and sets up that region so that it will be shared with any other process making the same call to `$ShareMemory`.

The section name is equatable, making it easy to run multiple copies of the same job without conflict. (For more information, type **Help Equate**.)

You may control whether the section name is local to your user ID or global to the system. If it is local (the default), then two users with different user IDs using the section name *flight_sim* will not share data.

Cachability and Flushing Registers

The smallest sharable unit on the System 6400 is a page (2048 bytes). The Datapool Assembler rounds up all the datapools to a page multiple in size.

To avoid stale data, `$ShareMemory` marks the shared pages as non-cachable. If you want to use shared cachable memory (because of its higher speed), place a call to `MM$CacheablePages` immediately after the call to `$ShareMemory`. Remember that on a multi-CPU system, you will have to manage cache flushing yourself by issuing calls to either `MM$FlushPages` or `OS$FlushCache`.

Datapool variables should not be cached in registers. FORTRAN marks all datapool variables as volatile. Pascal has a `{$v+}` option that marks variables as volatile. C has no such feature; we can only recommend that datapool variables not be register variables and that you not optimize the C code.

Initialized Data

The first process that calls `$ShareMemory` causes Embos to initialize the datapool to all zeros. No other facility is supplied for user initialization of a datapool. It is up to the user's program to initialize the data; this initialization *cannot* be done statically, but must be done at run-time. In particular, the FORTRAN DATA statement and the C and Pascal compile-time variable initialization constructs will be ignored.

Memory Management

CASE SENSITIVITY

All Embos languages, other than C, are case-insensitive; for example, the variable named "abc" is the same as the variable named "ABC." For linking-binding purposes, all external names, that is, those names that are known to the Binder, are shifted to uppercase. The DPasm follows this convention by making all external names uppercase. C, on the other hand, is a case-sensitive language. In C, the variable named "abc" is *not* the same as the variable named "ABC." If you are using datapools from C, you may either turn off the uppercase feature in the Datapool Assembler or you may declare the variables in the C program in uppercase characters: this is the recommended solution.

NAME SCOPING DATAPOL VARIABLES

If you are using multiple datapools within the same module/routine/process, you will want to make sure that there are no name conflicts between any two datapools. For example, if you use two datapools that each contain the variable "COUNT," how do you distinguish between the two variables? The DPasm has an option, called *SCOPENAMES*, that automatically prefixes the datapool name to each variable name (as in the case of the datapool descriptor name). For example, if *dir* is a variable in the datapool *sim*, the external name of *dir* will become *sim_dir*. The FORTRAN compiler has a *SCOPENAMES* switch so that it too can perform the same operation on datapool variable names. When this is the case, the user does not have to change any code. The default for both DPasm and FORTRAN is +*SCOPENAMES*.

SECTION NAMES

Global section names (that is, those names passed to \$ShareMemory) determine who else is allowed to share the specified memory. The programmer can control whether a section name is local to his or her user ID or global to the system. (The default is that section names are local to a user ID.) While they are not NameSpace entries, section names can be equated; so it is quite simple to handle multiple jobs simultaneously.

Memory Management

For example, suppose *SIMULATION* is an application consisting of 300 processes, each of which represents a part of an aircraft, uses a datapool called *PLANE*, and calls *\$ShareMemory* with the section name *PLANE*. The following command sequence, then, will run two detached copies of *SIMULATION*, each for a different aircraft with no conflict between them. No data will be shared between the two simulations even though it is shared by all the processes within each simulation. Equates are only necessary when multiple copies of an application are being run under the same user ID.

```
EQUATE plane plane1; RUN simulation +detached
EQUATE plane plane2; RUN simulation +detached
```

A process can use many different datapools simultaneously; of course, each must have its own unique section name.

NAMING CONVENTION

We recommend that you use the following naming convention: use the same name for the datapool name (specified by the *.datapool* statement in the *.dp* file), the section name (passed to *\$ShareMemory*), and the name of the *.dp* file (processed by *DPasm*). This is not required, but it makes it quite convenient and reduces the chance for confusion. In the sample program below, a realtime multi-user star-trek game uses a datapool called "trek," which is then assembled into *Trek.o* and shared using the section name "trek."

Memory Management

Here is a FORTRAN example of datapools. It consists of the following files:

Fortran source:

- strek.f
- global.f
- trekdatapool.f
- universe.f

Datapool source:

- trek.dp

Bind command:

- bind.strek

Here are the files:

```
c-----  
c FILE: universe.f  
c-----  
c Universe parameters. These describe the 3-  
c dimensional "size" of the universe. As you can  
c see, it looks remarkably like an et2510 terminal  
c screen.  
c-----  
      integer*4  coord1max,coord2max,coord3max  
      parameter (coord1max=20)  
      parameter (coord2max=80)  
      parameter (coord3max=1 )
```


Memory Management

```
c-----  
c FILE: global.f  
c-----  
c Global variables shared by all routines.  
c-----  
      common    // tag,me,tc,fmt  
+           ,pos_string,cs_string,home_string  
+           ,status,cmd,tcop  
      integer*4      me,tag,tc  
      character      cmd,tcop(2)  
      integer*4  
      fmt,pos_string(4),cs_string(4),home_string(4)  
      integer*4      status  
  
c-----  
c FILE: trekdatapool.f  
c-----  
c "Trek" datapool. This is the "full" datapool  
c declaration; it is included here just for  
c documentation's sake. Remember, with datapools  
c you need only declare those variables  
c actually used.  
c-----  
      parameter (max=100)  
      datapool /trek/ nextuser,alive,name  
+           ,coord1,coord2,coord3  
+           ,power,shields,lock  
      integer*4      power(max),shields(max),nextuser  
      integer*4      coord1(max),coord2(max),coord3(max)  
      logical*1      alive(max)  
      integer*8      lock  
      character*8     name(max)
```

Memory Management

```
c-----
c FILE: strek.f  (the main program)
c-----

    program strek
    implicit none
%include    'universe.f'
%include    'trekdatapool.f'
%include    'global.f'

c-----
c External functions.
c-----

    integer*4      $ShareMemory, $Stdout, $WhoAmI
    integer*4      $DatapoolAddress, DP$DatapoolSize

c-----
c Share the datapool. All cooperating programs have
c agreed to use the name "trek" for the global sec-
c tion name. The routine sets up this user's memory
c used by the trekdatapool to be shared. Note that
c the name "trek" may not be used by other users. In
c addition, datapool memory is set up as noncacheable
c to avoid "stale data." If you want to make it
c cachable, you can call MM$CacheablePages
c immediately after you call $ShareMemory.
c-----

    tag=$ShareMemory( %string 'trek'
+                      , %val
DP$DatapoolAddress(trek_dpdesc)
+                      , %val  DP$DatapoolSize(trek_dpdesc)
+                      )
```

Memory Management

```
c-----
c Get termcap info ("terminal capabilities") for
c cursor positioning, and so on. The variable tc
c holds the termcap information for your output
c device (terminal). If the output device is not a
c terminal but rather has been redirected into a
c file, don't bother with this.
c-----
      tc = 0
      call S$INIT ( pos_string )
      call S$INIT ( cs_string )
      call S$INIT ( home_string )
      call TERM$TCGetByFD (%val $stdout(), tc )
      if (tc .ne. 0) then
        call TERM$TCGetPaddedString (%val tc,'cl',cs_string)
        call TERM$TCGetPaddedString (%val
tc,'ho',home_string)
      end if

c-----
c Add this user to the datapool. For this operation,
c we must synchronize access to the datapool; lockIt
c and unlockIt handle this.
c-----
      call LockIt ( lock )
      me = nextuser
      nextuser = nextuser + 1
      call UnLockIt (lock )
```

Memory Management

```
c-----
c Initialize a new user: "me." This consists of
c giving me initial values for power and position,
c recording my name, and marking me as "alive." We
c choose the initial position by selecting a random
c location in the universe. (If you should be
c created on top of another user, it doesn't matter
c -- you can make up the rules.
c-----
      power(me)    = 10000
      shields(me)  = 10000
      coord1(me)   = ranint (coord1max)           ! row
      coord2(me)   = ranint (coord2max)           ! col
      coord3(me)   = ranint (coord3max)           ! not used
      call $WHOAMI ( %string name(me) )
      alive (me)   = .TRUE.

c-----
c Clear the screen and show everyone's position.
c-----
      call Home
      call Clear
      call UpdateScreen(me)

c-----
c Main loop. Read a single letter command from the
c screen's "command window" (it's on the next to
c last row), execute the command, and then
c update the screen.
c-----
10    call position (22,0)
      read (5, '(a1)') cmd
```

Memory Management

```
c-----
c Process and execute command.
c-----
      if ( cmd .eq. 'q' ) call quit ( me )
      if ( cmd .eq. 'u' ) call move ( me, -1,  0,  0)
      if ( cmd .eq. 'd' ) call move ( me,  1,  0,  0)
      if ( cmd .eq. 'l' ) call move ( me,  0, -1,  0)
      if ( cmd .eq. 'r' ) call move ( me,  0,  1,  0)
      if ( cmd .eq. 's' ) call updatescreen(me)
c-----
c Update the screen and read the next command.
c-----
      call updatescreen(me)
      goto 10
      stop
      end
c-----
c Lock the "trek" datapool. Spin loop on EXCH.OR
c until you get back a zero (meaning no one
c else has it locked.)
c-----
      Subroutine LockIt ( lock )
      integer*8 lock,oldlock
10    call OS$EXCHOR ( lock, %val 1, oldlock )
      IF (oldlock .ne. 0) goto 10
      end
c-----
c Unlock the trek datapool. The variable lock is
c marked as "volatile," which tells the compiler to
c force the value to memory and not optimize it.
c-----
      Subroutine      UnLockIt ( lock )
      integer*8      lock
      volatile        lock
      lock = 0
      end
```

Memory Management

```
c-----
c Update the screen and show me and all
c those Klingons.
c-----

      Subroutine      UpdateScreen(me)
      integer*4      me
      implicit none

c-----
c Trek datapool. Declare only those variables
c actually used.
c-----

      parameter (max=100)
      datapool /trek/ alive,coord1,coord2,coord3
      integer*4      coord1(max),coord2(max),coord3(max)
      logical*1      alive(max)

c-----
c Local variables
c-----

      integer*4      who

c-----
c Clear the screen.
c-----

      call Home
      call Clear

c-----
c Display Klingons.
c-----

      do 10 who=1,max
        if ( alive(who) .and. (who .ne. me) ) then
          call blip ( 'k', coord1(who), coord2(who),
                     coord3(who) )
        end if
10    continue
```

Memory Management

```
C-----
c Show me too.
C-----
      call Blip ( 'e', coord1(me), coord2(me),
        coord3(me) )
      end

C-----
c Move this user (me) around in the universe.
C-----
      Subroutine Move (me,x,y,z)
        integer*4 me,x,y,z
%include 'universe.f'

C-----
c "trek" datapool
C-----
      parameter (max=100)
      datapool /trek/coord1,coord2,coord3
      integer*4      coord1(max),coord2(max),coord3(max)

C-----
c Note that the universe is a sphere -- "rolling"
c off one end brings us around to the other.
C-----
      coord1(me) = MyMOD ( (coord1(me)+x), coord1max )
      coord2(me) = MyMOD ( (coord2(me)+y), coord2max )
      coord3(me) = MyMOD ( (coord3(me)+z), coord3max )
      end
```

Memory Management

```
c-----
c MyMOD. Handles negative integers correctly.
c-----
      INTEGER*4 function  MyMod (i,j)
      integer*4 i,j
      if (i .ge. 0) then
        MyMod = MOD (i,j)
      else
        MyMod = j - MOD(-i,j)
      end if
      end

c-----
c Blip. Display the character at the specified
c position.
c-----
      Subroutine Blip (ch,x,y,z)
      character ch
      integer*4 x,y,z
%include 'global.f'
      if (tc.ne.0) then
        call position(x,y)
        print *,ch
      end if
      end

c-----
c Position the cursor.
c-----
      subroutine      position(row,col)
      integer*4      row,col
%include 'global.f'
      if (tc .ne. 0) then
        call Term$TCfmtCursorPos
+         (%val tc, pos_string, %val row, %val
           col )
        call $PutStr ( pos_string )
      end if
      end
```


Memory Management

```
C-----
C Home the cursor.
C-----

      subroutine      home
%include 'global.f'
      if (tc .ne. 0) then
          call $PutStr ( home_string )
      end if
      end

C-----
C Clear the screen.
C-----

      subroutine      clear
      character       ch
%include 'global.f'
      if (tc .ne. 0) then
          call $PutStr ( cs_string )
      end if
      end

C-----
C Return a random integer between 1 and max.
C-----

      integer function      RanInt ( max )
      integer*4      max
      integer*8      OS$READREALTIMER
      integer*8      val
      val = OS$READREALTIMER
      if (val .lt. 0) val=-val
      val = mod(val,max)
      if (val .eq. 0) val=1
      RanInt = val
      end
```

Memory Management

```
c-----
c Quit. That's all folks.
c-----
      subroutine      quit ( me )
      integer*4       me
      call die ( me )
      stop
      end

c-----
c Die. Argggg!
c-----
      subroutine      die ( me )
      integer*4       me

c-----
c Only need to set the "alive" field in
c the datapool.
c-----
      parameter      (max=100)
      datapool       /trek/alive
      logical*1      alive(max)
      alive(me)      = .FALSE.
      end
```

Memory Management

```
c-----  
c FILE: trek.dp  
c-----  
.datapool trek;  
.what '@(#) trek 1.1 870721 ~';  
  
sec1 = trek + 0;  
sec2 = sec1 + 16;  
sec3 = sec2 + 1500;  
sec4 = sec3 + 1200;  
  
lock      = sec1 + 0      i8      'trek table lock '  
;  
nextuser  = sec1 + 8      i4      'next free user ID'  
;  
  
alive     = sec2 + 0 100i11 'user alive? '  
;  
name      = sec2 + 100 100c8 'user name'  
;  
  
coord1    = sec3 + 0 100i4 'position coordinate 1'  
;  
coord2    = sec3 + 400 100i4 'position coordinate 2'  
;  
coord3    = sec3 + 800 100i4 'position coordinate 3'  
;  
  
power     = sec4 + 0 100i4 'power level '  
;  
shields   = sec4 + 100 100i4 'shield level'  
;  
  
.end ;
```

Memory Management

```
C-----  
c FILE: bind.strek  
C-----  
    declare dp /embos/public/library/dplib  
    bind strek,trek +sf lf=[dp]
```

Now run the following command sequence:

DPasm trek

Fortran strek

Bind trek

Once this is done, every person who uses Strek will share the Trek datapool and become a participant in a game in which varying numbers of people can participate.

MAPPED-FILE ACCESS

The two basic types of file access are mapped-file access and access to data through an access manager or UNIX kernel. The fundamental differences are that mapped files only apply to disk files and when accessing data through an access manager or kernel, there is an additional process (other than the application itself) that retrieves and updates the data.

When using mapped-file access, the data on disk is mapped into the application's virtual address space rather than passed through a separate process. Thus, by merely reading and writing memory locations, the file itself is read and written to. Mapped-file access relies upon the mechanisms of the Memory Manager to bring the data into memory when needed and write it out to disk when it is no longer needed. It is also possible to use Memory Manager services to make sure that the data written to a file open for mapped-file access is actually written to disk.

The principal advantage of mapped-file access is that it is significantly faster than going through an access manager. One reason one would choose to access data through an access manager is that it provides concurrency control, that is, allows a process to share file access by multiple processes. Files within the UNIX file system can only be accessed through the UNIX kernel. However, System Foundation files (or entire UNIX file systems) can be mapped into UNIX processes.

If you have multiple processes writing to a file, you must use an access manager. If you have multiple processes reading a file, it is always safe to map the file. If you have one process writing to the file and many processes reading the file, the reading processes may either not get the latest copy of the data or not get all of the data. If you want to make sure that all the sharing processes use the same data, use an access manager to access the data.

In most types of Embos and EMS disk files, in addition to the data itself, there are various fields of control information. The structure of the data is normally known and controlled by the access manager, so that when doing file access through an access manager, the access manager interprets the control information and passes only the necessary data to the application.

In the case of mapped-file access, there is no access manager to provide that service. However, the file system intrinsics can interpret the control information and take advantage of the performance gains provided by mapped-file access to open a file directly. When an application opens an Embos or EMS disk file for exclusive access, the file system intrinsics automatically open that file in Mapped mode (to avoid having to use the access manager).¹⁰ Then the application can call the same file system intrinsics (such as FS\$Read and FS\$Write) that it would call when going through the access manager, but with the performance benefits of mapped-file access.

10 The file system intrinsics contain much of the same logic that access managers do.

Memory Management

RAW-MAPPED FILES

Both the access manager and the File System Intrinsic use the same file-access method. A file created by raw-mapping is an unstructured file; it is just an array of bytes and is usually not readable by a standard access manager. In particular, once opened and raw-mapped, you cannot use file system intrinsics such as `FS$Read` or `FS$Write` to access the file. Since the file looks like memory (and is, in fact), your process accesses it as if it were part of your virtual address space (which it is).

A process can only access a raw-mapped file as an unstructured file. The application takes full responsibility for managing not only the data in the file but also the control information (if any).

Raw-mapped files are best used for applications in which high speed is a critical factor or applications that must be able to copy a data structure directly from memory to disk. When you want to retrieve information from one or more external devices and build it into a form you can save, you can raw-map the file under the data structure in such a way that the data structure can be forced to disk as you build it.

The application is free to put data of any type and control structure into a raw-mapped file. Also, the data is not copied between a Map Window and a data buffer; the data is directly accessible by the application. A potential use of a raw-mapped file in realtime applications is to use this as a means of checkpointing vital data structures that are maintained in memory. For example, when a realtime process starts up, it can open a file and raw-map the file into its virtual address space. In that same address space, the realtime process can build the data structures it may need to checkpoint while the process is running. Using memory management services, the realtime process can then periodically force those data structures to disk.

Applications that gather information from external devices, such as sensors that transmit data at a high rate of speed, can also use raw-mapped files. The application can open a raw-mapped file, and write the data it receives from the devices into memory, allowing the Memory Manager to write the data to disk. Some applications maintain

Memory Management

a different type of data structure in their memory than on disk. For example, a linked list structure is common in memory but sequential record access is more common on disk files. Applications that can use the same type of data structure in memory as on disk would find raw-mapped files useful.

As a final example, the Profiler raw maps the profile file into an application's address space and keeps its tables there. In this way, if the application terminates, all the Profiler's data structures are saved on disk. "ProfileInfo" then raw maps that same file, analyzes the data, and reports the results.

CHAPTER 5

THE MESSAGE SYSTEM

INTRODUCTION

This chapter is directed toward programmers who wish to use the System 6400 message system. It discusses message system concepts and provides examples of how to use the message system to solve various problems. Most examples are in Pascal; FORTRAN and C programmers should refer to the sections on using FORTRAN and C at the end of this chapter.

Most programmers do not need to use the message system directly; processes can communicate quite well through high-level mechanisms such as pipes. Programming the message system is useful, however, when you need the extra performance it provides or when you are writing a system or low-level service, such as a server, access manager, or device driver.

LINKS, FUNNELS, AND CHANNELS

Links and funnels are the basic elements of the message system: messages are sent on links and received on funnels. Each link and each funnel is owned by exactly one process -- each process can have up to 65,535 links and 255 funnels. Each link points into exactly one funnel, but a single funnel can have many links pointing into it. A link can point into a funnel in the same process or another process. A link pointing into a funnel that belongs to the same process is called a

Message System

self-link, which is an important item in setting up process-to-process communication paths (this is discussed in more detail later in this chapter).

Most links point into funnels in other processes, and are used to send messages to those processes. Since each link points into exactly one funnel, it also points to exactly one process. (See Figure 5-1.) Links can be copied and passed from one process to another, but each copy of a link still points into the same funnel as the original. The funnel into which a link points cannot be changed.

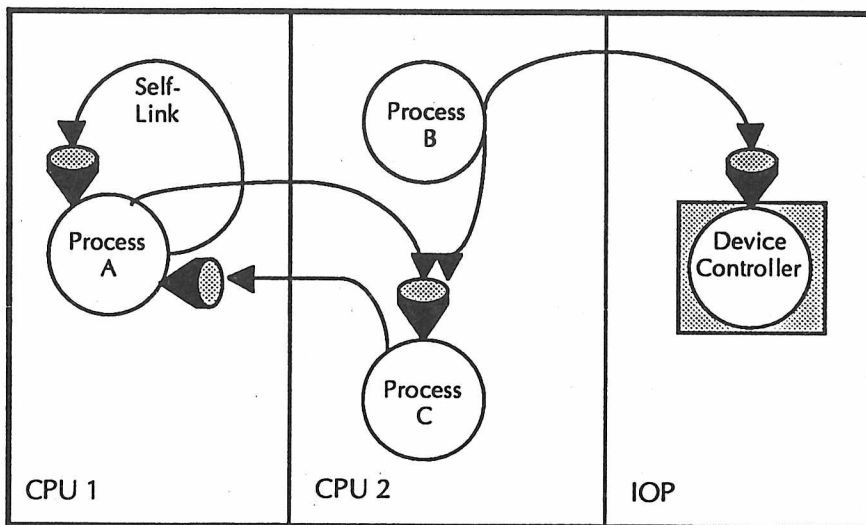


Figure 5-1. Links, Funnels, and Processes

The links for a process are kept in a link table, and the funnels in a funnel table. Link IDs and funnel IDs are indexes that select a particular entry in the table. These tables cannot be modified or accessed directly, but a process can read individual entries in its link and funnel tables with the `MS$ReadLTE` and `MS$ReadFTE` intrinsics. (Under UNIX, `e_read_lte` and `e_read_fte` respectively.) A link table entry, for example, contains information such as the process ID and funnel number into which the link points.

Channels provide a way to group funnels; they also control message priorities and the reception of messages and interrupts. Each funnel can be assigned to one of sixteen channels. By default, all new funnels are assigned to Channel 15 (the lowest priority). All sixteen channels are created when a process is spawned and are never deleted during the life of the process. The sixteen channels correspond to the sixteen local process priorities.

CREATING AND DELETING FUNNELS

The `MS$CreateFunnel` intrinsic creates funnels; the `MS$DeleteFunnel` intrinsic¹ deletes funnels. (UNIX: `e_del_funl`) Funnels cannot be copied or passed. You can call the `MS$CreateFunnel` intrinsic (UNIX: `e_cr_fun`) in the following manner:

```
funnelID := 0; MS$CreateFunnel( funnelID )
```

The newly created funnel ID is returned in `funnelID`. Be sure to initialize it to zero before the call.

To delete a funnel, call `MS$DeleteFunnel`. When a funnel is deleted, all messages waiting on that funnel (that is, messages sent to that funnel that have not been received) are discarded, and any links sent with those messages are deleted. (`MS$DeleteFunnel` handles this automatically, but before calling `e_del_fun`, all messages must be explicitly deleted.) However, links into the deleted funnel that are held by other processes are not deleted, since they are defined in the link tables of their respective processes.

Any attempt to send a message into a deleted funnel on one of these links results in an error, but if the funnel is subsequently recreated, the links again become valid links. Therefore, a funnel should *not* be deleted unless all the links pointing into it have first been deleted. Most

¹ To accommodate normal line lengths, long intrinsic names will occasionally be broken at the end of a line (as in this case). However, please note that hyphens are never part of intrinsic names.

Message System

applications avoid this problem by creating all the required funnels during initialization and never deleting them.

An alternative to deleting a funnel is to disable it temporarily with `MS$DisableFunnel`. You can enable the funnel again with the `MS$EnableFunnel` intrinsic. Any process that tries to send a message to a disabled funnel receives an `MS$FunnelNotEnabled` status, but messages already sent to the funnel are not affected and can still be received.

CREATING AND DELETING LINKS

A process can acquire links in one of two ways: receiving them from another process (discussed later in this chapter under “Receiving Messages”), or creating them with the `MS$CreateLink` (or `e_crt_link`) intrinsic. `MS$CreateLink` creates a self-link, which is a link that points into one of a process's own funnels. A process usually copies or passes self-links to other processes so that they can send messages to it.

To create a link, fill in a parameter block of type `MS$CreateLinkParamType` (UNIX: `cl_param`).² The fields in this block are as follows:

- *funnelID*. A one-byte unsigned integer that defines the funnel into which the newly-created link will point. This must be the ID of a funnel that has already been created.
- *linkID*. A two-byte unsigned integer that returns the ID of the newly-created link. This field must be initialized to zero before calling `MS$CreateLink`.
- *linkCode*. A two-byte unsigned integer of arbitrary value. The `LinkCode` helps to identify a message when it is received. (See “Receiving Messages” below.)
- *linkRights*. An 8-bit bit mask that defines whether the link can be passed or copied, and whether a notification message

² For the format of this and other message system parameter blocks, refer to these files: Embos and EMS: `/embos/include/types/MS$RecordTypes.p`
BSD: `/user/include/machine/emsg.h` System V: `/user/include/sys/emsg.h`

should be sent when the link is copied, passed, or deleted. (See "Notifications and Rights" below.)

- *linkGrail*. An array of 48 bits the holder of the link uses to help identify it. The LinkGrail cannot be changed, but remains the same when the link is copied or passed, and can be examined by the holder of the link. The LinkGrail is rarely used by application programs, and is normally set to all zeroes.

Once the link is created, all the various parameters are fixed. If a process copies or passes the link to another process, the link retains all these properties, though the link IDs of the various copies may be different. Remember that the link ID is actually the index into the link table; all the other information is what is in that link table entry.

To delete a link, use the `e_del_link` library routine or the `MS$Delete-Link` intrinsic.

SENDING MESSAGES

A message consists of a message parameter block, followed by an arbitrary sequence of data, with a data length of up to 888 bytes. The parameter block defines which link is to be used to send the message, and may also identify a second link to be copied or passed with the message. The size and format of the parameter block depends on which of the Send routines is used to send the message, as explained more fully in the following sections. For a message that contains a link (that is, a copy-link or pass-link message), the maximum data length is 872 bytes.

When a message is sent, the message system verifies that the links are defined, that the receiving process is alive, and that the funnel into which the message is to be sent is not disabled or deleted. Then the message data is copied from the sender's address space into *system message buffers*³ and queued on the funnel of the receiving process. At

³ The system message buffers are a separate area of memory reserved for the message system.

Message System

this point, the message has been “delivered,” the Send instruction is finished, and any modifications to the sender's original data buffer does not affect the message.

Once a message has been delivered, it remains queued on the receiver's funnel until the receiver does a Receive by calling `MS$Receive`, `e_rcv`, or one of their variants. If the receiving process is already waiting for the message, it is awakened immediately. Or, if the funnel is an interrupting funnel, the message is received immediately and the interrupt routine defined for that funnel is executed (unless higher-priority interrupts are pending). When the message is received, it is copied from the system message buffers into the receiver's address space.

For programming convenience, the parameter block and message data can either be contained in a single buffer or arbitrarily split into two separate buffers. Breaking the message into two parts has no effect on its contents; the receiving process cannot tell how the sender divided the message. The length of the parameter block is not included in the total message length. For example, if you have single buffer called *parmsAndMsg* that contains both the Send Parameter block and a message that is ten bytes long, you would call `MS$Send` (UNIX: `e_send_msg`) like this:

```
MS$Send(  {addr1=} adrord(paramAndMsg),  {len1=} 10,  
          {addr2=} 0,                    {len2=} 0 );
```

In some applications, it may be more convenient to maintain a single buffer that contains the Send parameter block, along with multiple data buffers. For example, if you have one buffer named *sendParm* and another buffer called *msgBuf* that contains a ten-byte message, you would call `MS$Send` like this:

Message System

```
MS$Send( {addr1=} adrord(sendParm), {len1=} 0,  
          addr2=} adrord(msg),      {len2=} 10 );
```

Many messages consist of a small number of fixed-length fields (such as a message code, reply code, data length, and time stamp) followed by a variable amount of data. These fixed-length fields are known as a *message header*. In these cases, you can avoid copying the data an extra time by maintaining a separate buffer that contains the Send parameter block followed by the message header.

For example, let us presume that the message header is six bytes long and that it follows the Send parameter block in a buffer called *SendParms&Header*. To send 500 bytes of data already in a buffer called *dataBuf*, you would call MS\$Send like this:

```
MS$Send( {Addr1=} adrord(sendParms&Header), {len1=} 6,  
          {Addr2=} adrord(dataBuf),          {len2=} 500 );
```

Message System

SEND ROUTINES

There are three Send routines: `MS$Send`, `MS$CopyLink`, and `MS$PassLink`. Their UNIX equivalents are `e_send_msg`, `e_copy_link`, and `e_pass_link`. While they all send messages, `MS$CopyLink` and `MS$PassLink` also allow a link to be sent along with the message. The contained link cannot be received as data by the receiver of the message; instead, it is placed directly into the receiver's link table when the message is received.

The difference between `MS$PassLink` and `MS$CopyLink` is that after the latter, the sender still has a copy of the link. After an `MS$PassLink`, however, the link is passed to the receiver and is not retained by the sender, so any attempt by the sender to use the link after it has been passed will fail. `CopyLink` and `PassLink` allow up to 872 bytes of data to be sent in a message, while `Send` allows up to 888 bytes.

`MS$Send` requires an eight-byte parameter block, as defined by the Pascal record `MS$SendParamType` (UNIX: `sn_param`).⁴ The only field that must be filled in is the *Link ID* field, which is a two-byte unsigned integer that defines the link used to send the message. The format of the parameter block is as follows:

- Bytes 1-2: Unused, set to zero
- Bytes 3-4: Link ID
- Bytes 5-8: Unused, set to zero

`MS$CopyLink` and `MS$PassLink` use the eight-byte parameter block defined by the Pascal record `MS$CopyPassLinkParamType` (under UNIX: `pl_param`). The fields are as follows:

- *LinkID*. A two-byte unsigned integer that defines the link used to send the message.

⁴ Refer to `/embos/include/types/MS$RecordTypes.p` to see the format of this parameter block. UNIX: `/usr/include/[syslmachine]/emsg.h`

Message System

- *LinkIDtoCopyPass*. A two-byte unsigned integer that defines the link to be copied or passed with the message.

The format of the parameter block is as follows:

Bytes 1-2: Unused, set to zero
Bytes 3-4: LinkID
Bytes 5-6: Unused, set to zero
Bytes 7-8: LinkIDtoCopyPass

For example, suppose an application has a link to a server process called *servLink*. Let's assume you want to give a copy of that link to a process to whom you have a link called *friendLink*. You also want to include the integer 1234 in the message, which the "friendly process" will know means that you are sending it a link to the server. The procedure to accomplish this follows:

```
procedure sendLinkToFriend;
var
    copyParam    : MS$CopyPassLinkParamType;
    msgCode      : integer := 1234;
begin
    copyParam.linkID           := friendLink;
    copyParam.linkIDtoCopyPass := servLink;
    MS$CopyLink( adrord(copyParam), 0, adrord(msgCode), 4 );
end;
```

NOTIFICATIONS AND RIGHTS

A link may have three kinds of notification attributes associated with it: *delete*, *copy* and *pass*. These notification attributes are defined when the link is created and cannot be changed. When a process enables a notification attribute, the message system sends a notification message down the link at the time the corresponding message system operation is done. When the process holding the link terminates without first deleting the link, a delete link notification message is also sent. A notification message cannot be disabled or forged. Therefore, a process can always tell how many copies of a link are outstanding by counting the number of times the link has been copied and deleted.

For example, suppose a process has a link called *servLink* to a server process with its copy notification enabled. If the process gives a copy of *servLink* to another process with *MS\$CopyLink*, a message is sent to the server process via *servLink* telling the server process that the link has been copied.

A delete link notification message consists of zero bytes of data, while a copy and pass link notification message consists of a four-byte message, the first two bytes of which contain the process ID of the process to which the link was copied or passed. The *MessageType* field in the parameter block received with the message defines which type of link notification message was sent. (Refer to "Receiving Messages" below for more information on the *MessageType* field.)

A link can have two kinds of rights: *copy* and *pass*. These rights control whether any other process (besides the original creator of the link) can copy or pass the link. If Process A receives a link from Process B, it must have pass rights for Process A to call *MS\$PassLink* with it as the link ID to *CopyPass*; the same is true for copy rights and *MS\$CopyLink*. This means that a link with neither copy nor pass rights can only be held by the processes that received it directly from the link's creator.

RECEIVING MESSAGES

Messages are received with one of the following four intrinsics:

- `MS$Receive` (UNIX: `e_rcv`)
- `MS$ReceiveLink` (`e_rcv_link`)
- `MS$ReceiveChannels` (`e_rcv_chans`)
- `MS$ReceiveLinkOnChannels` (`e_rcv_link_on_chans`)

The first two receive the first waiting message on a specific funnel, the latter two receive a waiting message from a funnel attached to a set of channels specified in the call. For example, a process can receive the first message on any funnel attached to Channels 10, 12, or 15. Messages on lower-numbered (thus, superior priority) channels are chosen before those on higher-numbered channels.

The only way to receive a message that contains a link is to use `MS$ReceiveLink` or `MS$ReceiveLinkOnChannels`. For this reason, these two routines are more general than `MS$Receive` and `MS$ReceiveChannels`. If used to receive a message that does not contain a link, they receive the message successfully, but return a `MS$MessageDoesNotContainLink` status (UNIX: `MSYS_NOLINK`).⁵

`MS$Receive` and `MS$ReceiveChannels` are used to receive messages that do not contain links. If used to receive a message that does contain a link, they return the data portion of the message along with a `MS$MessageContainsLink` status (UNIX: `MSYS_HASLINK`), but the entire message waits on the funnel; the next receive on that funnel sees the same message again. Table 5-1 summarizes the pertinent information on these four intrinsics:

5 The file `/embos/include/status/MS$Status.p` holds the definitions for these status values. Under UNIX, `.../emsg.h` contains status values.

Message System

Message Intrinsic	Receive Link	Funnel Category
MS\$Receive	No	Specific
MS\$ReceiveLink	Yes	Specific
MS\$ReceiveOnChannels	No	Group
MS\$ReceiveLinkOnChannels	Yes	Group

Table 5-1. The Receive Ininsics

When a message is received, it consists of a parameter block followed by the message data. The data part of a message cannot exceed 888 bytes. The receive parameter block is not the same as the parameter block used to send the message. Instead, the receive parameter block contains fields that define the type of Receive to be done and returns values that identify the message by its link code, message type, sending process ID, the funnel on which the message was received, and the length of the message. The size and format of the parameter blocks vary, depending on which Receive intrinsic is used. Use the MS\$ReceiveParamType (UNIX: rv_param) parameter block for MS\$ReceiveLink and MS\$Receive. Use the MS\$ReceiveOnChannelsParamType (UNIX: ro_param) parameter block for MS\$ReceiveChannels and MS\$ReceiveLinkOnChannels.

As with the Send intrinsics, the received message may be divided into two separate buffers, allowing the parameter block (and perhaps user-defined control information) to be received separately from the message data. The received message is placed into the two buffers, filling buffer 1 first. As long as the two buffers are a combined size of 888 bytes or greater, the entire message can be received. (Note that it is assumed that buffer 1 has additional space at the beginning for the receive parameter block.)

However, if the buffers are not large enough to hold the message, as much as will fit is copied into the buffers and the status MS\$Message-TooLong (UNIX: MSYS_2LONG) is returned. In this case, the whole

message remains on the funnel and must be received again into a larger buffer. For this reason, many applications simply allocate a buffer of 888 bytes for receiving messages. (A receive parameter block must precede this buffer, or provide the block as buffer 1 and define the 888-byte buffer as buffer 2.)

Both the `MS$Receive` (UNIX: `rv_param`) and `MS$ReceiveLink` intrinsics use the `MS$ReceiveParamType` parameter block. The fields of this block are divided into input and output fields. They are defined as follows:

INPUT FIELDS

- *FunnelID*. A one-byte unsigned integer that defines the funnel on which the message should be received.
- *TypeOfReceive*. An 8-bit bit mask that defines various options on the Receive operation. Specifying `MS$Interrogate` (under UNIX, it is `RT_INTERROGATE`) leaves the message on the funnel after the Receive is complete. If you specify `MS$Synchronous` (`RT_SYNCHRONOUS`) and there are no messages on the funnel, the Receive waits until a message is received. If you do not specify `MS$Synchronous`, the Receive returns immediately with a status of `MS$NoMessageInFunnel` (UNIX: `MSYS_NOMSG`). Another option, `MS$Dismiss` (`RT_DISMISS`), is discussed under "Interrupts and Priorities" later in this chapter.
- *PreferredLinkID*. A two-byte unsigned integer that defines which link ID should be used if a link is received. Normally it is initialized to zero, in which case the next available link ID is used. For `MS$ReceiveLink`, the link ID of the received link (if any) is returned in this field when the Receive completes. If your process calls `MS$ReceiveLink` in a loop, be sure to set this field to zero before each call.

Message System

OUTPUT FIELDS

The *ControlInfo* record is a series of fields in the Receive Parameter block that return identifying information about the message when it is received. The fields in this record are set after the message has been received. These fields are defined below:

- *LinkCode*. A two-byte unsigned integer that contains the link code of the link on which the message was sent.
- *FunnelID*. A one-byte unsigned integer that contains the funnel on which the message was received. This is useful when either *MS\$ReceiveChannels* or *MS\$ReceiveLinkOnChannels* were called since the message can be received on any of several funnels. The values for these two fields are always the same as the Receive Funnel ID.
- *MessageType*. An eight-bit bit mask that defines the type of message sent. If the message includes a link, *MS\$IncludesLink* will be set (UNIX: *MT_L_INCLUDED*). If the message is a link notification, *MS\$LinkCopied*, *MS\$LinkPassed*, or *MS\$LinkDeleted* will be set. The UNIX equivalents are *MT_L_COPIED*, *MT_L_PASSED*, and *MT_L_DELETED*.
- *FromProcess*. A 2-byte unsigned integer that contains the process ID of the sending process.
- *NumBytes*. A 2-byte unsigned integer that contains the number of data bytes in the message, but it does *not* include the parameter block. If the buffers supplied to the Receive intrinsic are too small to receive the message, this number represents the minimum buffer size necessary to successfully receive the message.

Table 5-2 shows the format of the MS\$ReceiveParamType parameter block.

Values Set by Programmer	
Byte 1:	ReceiveFunnelID
Byte 2:	Unused, set to zero
Byte 3:	TypeofReceive
Byte 4:	Unused, set to zero
Bytes 5-6:	PreferredLinkID
Bytes 7-8:	Unused, set to zero

Receive Control Information	
Bytes 9-10:	Link Code
Byte 11:	Funnel ID
Byte 12:	MessageType
Bytes 13-14:	FromProcess
Bytes 15-16:	NumBytes

Table 5-2. The ReceiveParamType Parameter Block

Message System

For example, to receive a message on funnel *fun* into buffer *buf*, you could use the following procedure:

```
var
  rcvParam : MS$ReceiveParamType;
  buf      : array[1..MS$MaxMessageLength] of char;
  rcvParam.typeOfReceive    := [MS$Synchronous];
  rcvParam.funnelID         := fun;
  MS$Receive( adrord(rcvParam), 0, adrord(buf),
    MS$MaxMessageLength );
```

The `MS$ReceiveChannels` and `MS$ReceiveLinkOnChannels` intrinsics use the `MS$ReceiveOnChannelsParamType` parameter block. It is identical to the `MS$ReceiveParamType` parameter block, except that the `FunnelID` field is replaced by a `ChannelMask` field.

- The `ChannelMask` field is a 16-bit bit mask with one bit per message-system channel. Setting a bit in the `ChannelMask` allows messages to be received on the corresponding channel.

Table 5-3 shows the format of the MSSReceiveOnChannelsParamType parameter block.

Values Set by Programmer	
Bytes 1-2:	ChannelMask
Byte 3:	TypeOfReceive
Byte 4:	Unused, set to zero
Bytes 5-6:	PreferredLinkID
Bytes 7-8:	Unused, set to zero

Receive Control Information	
Bytes 9-10:	LinkCode
Byte 11:	FunnelID
Byte 12:	MessageType
Bytes 13-14:	FromProcess
Bytes 15-16:	NumBytes

Table 5-3. The Receive on Channels Parameter Block

Message System

DELETING MESSAGES

Messages that do not contain links can be deleted from a funnel without receiving them by calling `MS$DeleteMessage` (Under UNIX: `e_del_msg`). If the message contains a link, this intrinsic will fail and return an `MS$MessageContainsLink` status (UNIX: `MSYS_HASLINK`). The only way to delete such a message is to receive it (successfully) and then explicitly delete the link, as shown below:

```
var
  fun          : MS$FunnelIDtype;
  buf          : record
    rcvParam   : MS$ReceiveParamType;
    space      : array[1..MS$MaxMessageLength] of char;
  end;
{...}
buf.rcvParam.typeOfReceive      := [MS$Synchronous];
buf.rcvParam.funnelID          := fun;
buf.rcvParam.preferredLinkID    := 0;
MS$ReceiveLink( adrord(buf), MS$MaxMessageLength, 0, 0 );
MS$DeleteLink( buf.rcvParam.preferredLinkID );
```

INTERRUPTS AND PRIORITIES

LOCAL PRIORITIES

There are two kinds of priorities: *global* and *local*. The 256 global priorities are divided into one of the four execution priorities: realtime, timesharing, batch, and background. (See "Global and Local Priorities" in Chapter 3.) The System 6400 uses global priorities to determine which process will run next. Local priorities (of which there are sixteen) have meaning only within a process; they affect when interrupts in the process are accepted and the order in which messages are received. The channels correspond to local priorities, with 0 being the highest priority and 15 the lowest.

For realtime processes, there is a direct correspondence between local priority and global priority; for other kinds of processes, the System Foundation manages the global priorities for processes running at inferior execution classes. The local-to-global priority map for realtime processes is maintained by the Process Manager. Under Embos and EMS, it is initialized from the program's Bound file, and can be set either with the **ChannelPri** command (see the Help file) or, at run-time, with the **Pro\$ResetPriority** intrinsic. Under UNIX, the channel priority must be set explicitly with **Pro\$ResetPriority**.

A process's local priority can be changed by four events:

1. Delivery of a message
2. Processing an interrupt
3. A call to **MS\$SetLocalPriority** (or **e_set_local_pri**)
4. Specifying the **MS\$Dismiss** option (or **RT_DISMISS**) on a Receive

A message is considered to be delivered when it is sent, not when it is received, which may be much later. When a message is delivered to a process, the priority of the process is raised to that of the highest priority active channel, if it is not already at or above that priority. An *active channel* is one with messages waiting to be received. The highest priority active channel is the active channel with the lowest channel number. Message delivery does not lower the priority of a process.

Interrupts also affect the priority of a process. When an interrupt occurs, the current local priority is saved on the stack. When interrupt processing is complete, the process exits from the interrupt with a call to **OS\$Exit** (or by executing the **EXIT** instruction). This causes the local priority to be set to the greater of the saved local priority and the highest priority active channel. Thus, interrupt processing can raise, but not lower, the priority of a process.

Calling **MS\$SetLocalPriority(channel)** changes the priority to that of the specified channel, or to that of the highest priority active channel, whichever is higher. Setting the **MS\$Dismiss** option on a Receive instruction is equivalent to calling **MS\$SetLocalPriority(15)** immediately

Message System

before the Receive. The result is that the priority will be set to the same local priority as the channel on which the message is received. Otherwise, a low-priority message received after a high-priority message would not cause the priority to be reduced. The only ways to lower a process's priority are to set the Dismiss option on a Receive and call `MS$SetLocalPriority`.

In addition to handling interrupts, local priorities also manipulate the global priorities of realtime processes. For example, let's assume you have a process that handles two kinds of requests called "green" and "red." Green requests are low priority and red requests are high priority, so your process's execution priority should be adjusted accordingly. Assume red requests come in on `RedFun` and green requests on `GreenFun`.

1. Attach `GreenFun` to Channel 14 and `RedFun` to Channel 13.
2. Then set your channel priority map in the Bound file (or make an explicit call to `Pro$ResetPriority`) to give Channel 14 a low global priority and Channel 13 a high global priority.
3. Set the `MS$Dismiss` option in `MS$ReceiveLinkOnChannels`, which accepts a new request (on Channels 13 and 14, of course).

In this way, whenever a Red request is being processed or waiting, your process executes at a high priority; when only Green requests are being processed or waiting, the process executes at a lower priority.

AN INTERRUPTING FUNNEL AND CHANNEL

A channel can be designated as an interrupting channel by calling

- `MS$EnableChannelInterrupts(ChannelMask)` or
- `e_enable_chan_int(ChannelMask)`

where *ChannelMask* is a 16-bit bit mask. The correspondence between bits and channels is that the leftmost bit (that is, the bit with hex value 8000h) corresponds to Channel 0 and the rightmost bit (0001h) corresponds to Channel 15. Each channel whose bit is set to 1 in the mask

becomes an interrupting channel, while those channels whose bit is set to 0 are unaffected. The intrinsic returns the previous interrupt state of the channels in the ChannelMask.

The corresponding intrinsics, MS\$DisableChannelInterrupts or UNIX's `e_disable_chan_int`, disable interrupts on each channel whose bit is set to 1. The funnels attached to an interrupting channel are all interrupting, and should all have interrupt vectors set. The interrupt vectors are set with

- MS\$SetFunnelInterruptVector(funnelID, intVec) or
- `e_set_fun_int_vec(funnelID, intVec)`

where *intVec* is the address of the routine to call when a message is delivered on that funnel; the previous interrupt address is returned in *intVec*. An example follows.

```
var
  intFun      : MS$FunnelIDtype;
  intChan     : MS$ChannelIDtype;
  chanMask    : MS$ChannelMaskType := 16 of false;
  intAddr     : $AddressType;
{...}
MS$AttachFunnel( intFun, intChan );
intAddr := adrord( myInterruptRoutine );
MS$SetFunnelInterruptVector( intFun, intAddr );
chanMask[intChan] := true;
MS$EnableChannelInterrupts( chanMask );
```

INTERRUPT ROUTINES

When an interrupt occurs on an interrupting funnel, the information is pushed on the stack and control is transferred to the funnel's interrupt vector address. The interrupt procedure should receive or delete the message and perform an interrupt exit (which is *not* a normal procedure exit). The stack format is described in Chapter 4 of the *System Architecture* manual under "Instruction Set Composition."

Message System

EFFECT OF LOCAL PRIORITY ON INTERRUPTS

An interrupt to Channel *n* is accepted only if the current local priority is inferior (and therefore numerically higher) to the interrupting channel. This means that if your process is executing at local priority 7, interrupts on Channels 0 through 6 will be accepted. When your process exits the interrupt routine (with an OS\$IXIT call or the I\$EXIT instruction), the priority is set to the greater of the highest active channel and the priority the process was running at when the interrupt occurred.

Because delivery of a message on a channel raises a process's local priority, it is usually necessary to use superior priority channels for interrupts and inferior priority channels for ordinary messages. To see the problem, suppose a process has a normal funnel on Channel 5 and an interrupting funnel on Channel 7. The arrival of a message on Channel 5 raises the priority of that channel and prevents interrupts from occurring on Channel 7 until the local priority is lowered again.

MESSAGE SYSTEM CONSTRAINTS

Since messages that have been sent but not yet received are stored in system message buffers, there must be a mechanism that prevents a runaway sender (or lazy receiver) from consuming all the system message buffers. This mechanism is the *max attached* and *max in-transit limits* (both of which are imposed by the system). The max attached limit defines the maximum amount of unreceived data a process may have; the max in-transit limit defines the maximum amount of data a process may transmit without its being received. Thus, each message counts against the max in-transit limit of the sender and the max attached limit of the receiver.

When an MS\$Send would result in the sender's max in-transit limit being exceeded, the sender is blocked until some of its sent-but-unreceived messages are received by a process. The best solution is simply not to send large amounts of data without waiting for an acknowledgment.

PERFORMANCE CONSIDERATIONS

For most programs, the message system is so fast that the performance details are irrelevant. While the message system is very fast compared to other interprocess communication methods, it is slow compared to simple machine instructions: a process can typically execute over 250 instructions in the time it takes to transfer a 10-byte message.

The `MS$Send` and `MS$Receive` routines simply execute a single corresponding message system instruction and then handle the statstack. The extra procedure calls and statstack processing consumes about 16 microseconds per call; you can recover this time by writing the message processing routine in Assembly and coding the corresponding instructions in-line, but this is likely to be a small gain in most cases. The major cost of using the message system is the message system itself. Under UNIX (in C), the routines `e_send_msg`, `e_rcv`, and so forth can be automatically replaced with the corresponding instructions. For details, see `man e_ops`.

To send a message from one process to another residing on different CPUs, including the process switch and the receipt by the second process, takes approximately 100 microseconds plus an additional .45 microseconds per byte of message data. Sending a message from one process to another process residing on the same CPU takes approximately 130 microseconds (plus an additional .45 microseconds per byte of message data).

This time is about evenly divided between the sending and receiving processes, with the time to switch between them negligible -- about 10 microseconds if both are in register sets. This gives a transfer rate of slightly less than 2 Mbytes per second; clearly not the method of choice for moving large quantities of data. If you need to transfer large quantities of data between two (or more) processes, consider using the `MM$BidForSharedPages` and `MM$OfferSharedPages` intrinsics to share memory between the processes and using the message system only for synchronization. (See "Sharing Memory" in Chapter 4 and "Transferring Data" in Chapter 6.)

Message System

It may be appropriate to use the message system for data transfer if the volume of data is low (as in the case of terminal I/O) or there is a great deal of processing to be done on the data (as in the case of I/O to keyed files).

If realtime response to messages is critical, there are several other factors to consider. First, your process must be frozen in memory and locked into a register set. (See Chapter 4, "Memory Management.") Second, your process must run at super-realtime priority. If other processes are running on the same CPU at a higher priority, your process will wait. A super-realtime process that is locked into a register set and frozen in memory can respond to incoming messages within 50 microseconds.

PRIMITIVE MESSAGES

Primitive messages (PMSGs) are an alternate form of the general message system that can be employed for special circumstances in which both the speed of message delivery is critical and the amount of data to be transferred is very small.

The high speed of PMSGs is achieved by imposing a number of restrictions both on the messages and the way in which they are handled. The major restrictions are listed below.

- The size of a PMSG is fixed at 8 bytes, as opposed to 0 to 888 bytes for a standard message.
- PMSGs do not use funnels. Instead of the usual 256 funnels, there is a single *PMSG Receptor* attached to one of the message channels. All PMSGs received by a process are received through this single PMSG Receptor.
- PMSGs are not queued in message buffers, and therefore a PMSG may be lost if a second PMSG is sent to a process before the earlier one was received. This severely limits their use for most server processes, which cannot control the flow of requests to them.

Message System

- When a PMSG is received, none of the associated identifying information is available, such as the process ID of the sending process, the link code, funnel ID, and so forth.

PMSG RECEPTOR

The PMSG Receptor consists of a pair of Process Control Registers, which are part of the Process Control Block of each process. One register contains the data resulting from the latest PMSG, while the other register contains the following information:

- The address of the interrupt routine to be called when a PMSG is received
- A count of the number of PMSGs sent to the process since the last PMSG was received
- The channel number the PMSG receptor is assigned to
- The valid bit, which indicates whether the rest of the information in the register is valid

Because this information is maintained directly in the Process Control Block, it can be very quickly accessed by microcode.

PMSG DELIVERY

When a PMSG is sent, the microcode copies the data into the PMSG Receptor of the receiving process, and increments its PMSG counter. If an interrupt routine has been defined, it is called; otherwise the PMSG will be received when the next Receive instruction is executed. All of the overhead of allocating system message buffers and enqueueing them onto the proper funnel is eliminated.

FLOW CONTROL

Because PMSGs are not queued, the sending and receiving processes must cooperate to avoid losing messages. For example, if the sending process always waits for a reply before sending the next message, no

Message System

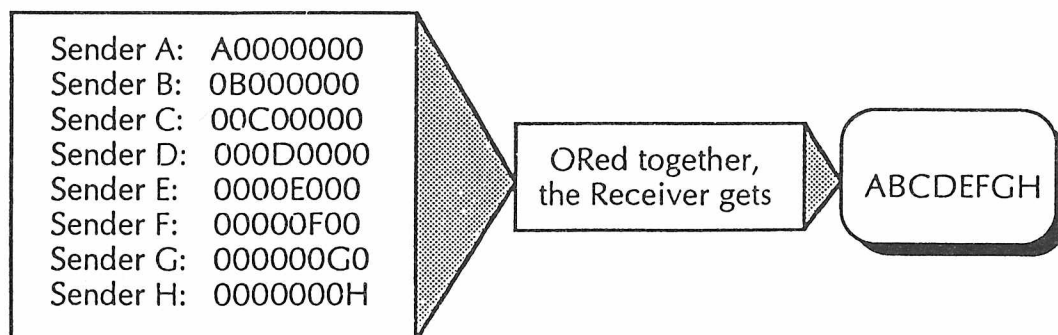
messages will be lost. Or the PMSGs may be sent at fixed intervals to initiate some action that is guaranteed to be completed within the interval.

When PMSGs simply signal some event in which the data is not important, flow control is not needed. Instead, the counter in the PMSG Receptor represents the number of such events that have occurred since the last PMSG was received. This counter is set to zero automatically each time a PMSG is received.

PMSGs SENT BY MULTIPLE PROCESSES

Because PMSGs are not queued, they are generally restricted to using a single sending process for each receiving process. However, a special feature of PMSGs extends their usefulness to multiple sending processes when the amount of data to be sent is less than 8 bytes. When a PMSG is delivered, the sending process can specify that the PMSG is to be ORed into the PMSG Receptor instead of overwriting the entire previous PMSG. If each sending process restricts its non-zero data to a separate field in the PMSG, PMSGs can be sent by multiple processes and ORed together without losing any data.

For example, suppose eight sending processes each want to send a byte to the same receiving process. If each sending process uses a different byte in the PMSG, all eight bytes can be received without losing data.



An arrangement such as the one above obviously requires careful co-ordination between the sending and receiving processes.

INTERPROCESS COMMUNICATION

Each process is restricted by hardware and microcode to accessing its own virtual address space. The only way a process may affect other processes or I/O devices is by sending and receiving messages. Because a process cannot destroy the address space of another process, bugs in one process cannot affect another process (except by sending erroneous messages). Errors do not spread because each process can check its incoming messages for accuracy.

There are two general methods by which two or more processes can communicate: the message system and shared memory.

What criteria do you use to determine which of these two methods is appropriate for your application? It is good practice to use messages for interprocess communication when you are passing pieces of information that are less than 888 bytes (which is the maximum size for a message) and the frequency of the communication is moderate. If you have large chunks of data to be shared or you have small chunks of data that need to be shared frequently, it is often a good idea to use shared memory instead of messages.

Unless memory is explicitly shared, one process cannot access the memory of another process. The system has no Supervisor mode. The message system enforces system security because the key to what a process can or cannot do depends on what links it has to other processes.

A key consideration in interprocess communication is the relationship between the processes: this relationship dictates which method of communication you should choose. There are three types of relationships between communicating processes:

- Spawned processes
- Forked processes

Message System

- A process from Job1 communicating with a process from Job2

When spawning or forking, the child process is created and the communication paths are set up automatically. When communicating from one job to another, both processes already exist and your process must establish a communication path by creating and opening a rendezvous.

When choosing between a spawn and a fork, the function of the child process is the most important consideration. If the child process is to perform a function completely different than its parent, you should spawn the process. A process fork, on the other hand, is a copy of the parent's code and unshared data. If the child process is going to perform a function that is similar or identical to the parent process (for example, two identical processes operating on a shared array at the same time), the method of choice is clearly a fork. You may also want to use a fork to save memory, since any shared data is not copied to the child process.

Another consideration in determining whether you want to fork or spawn a process is that a fork is usually much faster. The operating system has much less to do to implement a fork than it does for a spawn. Also, if the duration of the child process is going to be brief, the faster speed of a fork is significant. If the child process is going to run for a long time (minutes or longer), the set-up time is not important.

Why would you use a rendezvous instead of spawning or forking a process? You must use a rendezvous when you want to communicate between processes in different jobs. That is, use a rendezvous when a user process needs to communicate with an access manager or a system process, or when a user process from one job needs to communicate with a user process from another job. A rendezvous also provides the security of going through the NameSpace Manager so that you can provide access for users or processes that have the correct security and deny access for those who do not.

USING THE MESSAGE SYSTEM

Before two processes can communicate via the message system, they must first establish communication paths (that is, links) to each other. When one process forks or spawns the other process, these links are set up automatically. The child process uses its parent link (`MS$ParentLink`) to send messages to the parent process and its parent funnel (`MS$ParentFunnel`) to receive messages from the parent process.⁶ The parent process obtains the child link and funnel with the intrinsics `PM$GetLinkInPD` and `PM$GetFunnelInPD`. (See “Example Strategies” below.)

For two processes to establish communication via a rendezvous, the rendezvous must first be created and opened. The rendezvous can be created either by a command before the processes are running or by one of the two processes as they are running.

There are two types of rendezvous: *simple rendezvous* and *server rendezvous*. A simple rendezvous establishes direct communication between two processes. A server rendezvous allows many processes to establish communication with a single server process. (See “Server Rendezvous” later in this section.) In both cases, one process creates the rendezvous by calling `FS$CreateRZV` or `FS$CreateServerRZV`, respectively. Then the other process opens the rendezvous by calling `FS$Open`. Once the rendezvous is open, you can obtain the corresponding link and funnel by calling `FS$GetLinkInFD` and `FS$GetFunnelInFD`.

There is nothing to limit a forked or spawned process from using server rendezvous. But in most cases the extra work required to set up the communication paths would not be efficient.

⁶ The constants `MS$ParentLink` and `MS$ParentFunnel` are defined in the file `/embos/include/types/MS$Types.p`. You can find similar definitions under UNIX in `emsg.h`.

Message System

USING SHARED MEMORY

The second main method of interprocess communication is shared memory. There are three forms of shared memory: public-shared memory, private-shared memory, and fork-shared memory.

Public-shared memory consists of 64 Mbytes that is implicitly shared among all the applications on the system. Public-shared memory is easy to use because no set up is required. Every process on the system automatically has access to it. However, in most cases we recommend that you not use public-shared memory because there is no protection against other processes modifying this area of memory and there is no mechanism to dynamically allocate it.

Private-shared memory permits an application to share a portion of its address space with other processes. It is typically used between two functionally distinct processes (that is, process that are not the result of a fork). Private-shared memory is managed through the `MM$BidForSharedPages` and `MM$OfferSharedPages` intrinsics.

The advantage of private-shared memory is that access to the shared memory can be explicitly controlled (unlike public-shared memory). The major disadvantage of this form of shared memory is the set up required. The communicating processes must establish a message link to each other and use Memory Manager intrinsics to gain memory access to shared memory.

Fork-shared memory is used primarily (but not exclusively) by the `MT$` routines. Fork-shared memory is available to other processes that manage their environment themselves. For more information on each form of shared memory, see "Sharing Memory" in Chapter 4.

Like private-shared memory, access to fork-shared memory can be explicitly controlled. In addition, setting up fork-shared memory is easy. The parent process specifies which area of memory is to be shared before the child is forked. The child process can access the fork-shared memory immediately without any additional set up. The major disadvantage of fork-shared memory is that it can only be used with a forked process.

If the parent process shares memory prior to the fork, the child process will also have access to that memory. When using fork-shared memory, the virtual address space of an instruction resides in the same virtual address location for both processes. (When the memory for a forked process is not shared, the same virtual address points to two different addresses in physical memory.) At the time of the fork, the virtual addresses for the portions of memory that are shared point to the same physical address. This is managed internally by the Memory Manager.

When you determine that you want to use shared memory for inter-process communication, you need to also determine which form of shared memory to use. For a spawned process, use private-shared memory. For a forked process, use fork-shared memory.

When using private-shared memory for a spawn, you must use at least two calls. When using fork-shared memory to communicate between a parent and child in a fork, you need to only make one call from the parent process.

OBTAINING LINKS TO OTHER PROCESSES

There are two ways to obtain links to another process: receive them in messages or be born with them.

Standard Links and Funnels

Each process is created with certain predefined links and funnels. Except for the funnel for the Lifeline Interrupt Handler, you can delete or modify these links and funnels as you please. This is not recommended, however, because the system software depends on them and your program may behave strangely if you change these links and funnels. You will probably not need to use them directly; intrinsics will use them for you. In the list of links and funnels that follows, the link or funnel ID is given in parentheses after the name:

Message System

All Operating Systems

- *MS\$LifelineFunnel(1)*. Used by the system and attached to Channel 0. You cannot use or modify it in any way (nor can you modify Channel 0).
- *MS\$ExceptionFunnel(2)*. Attached to Channel 1; its interrupt handler is the system exception dispatcher. Any exception in your process results in a message into this funnel, where it is received and handled by the exception dispatcher. If you want to handle your own exceptions, we recommend you use intrinsics to install specific user exception handlers rather than replace the interrupt handler for the exception funnel.
- *MS\$ExceptionLink(1)*. Points into your own process's *MS\$ExceptionFunnel*. The microcode and software send exception messages down this link (such as overflow, range check high, and so forth.)
- *MS\$JobManagerLink(3)*. Points to your job manager, and is used for various job control functions.

Embos/EMS Only

- *MS\$ParentFunnel(3)*. Your parent process has a link into this funnel that it can use to send messages to your process. Your parent's link pointing into this funnel may or may not have delete, copy, or pass notification rights -- this is up to the parent process.
- *MS\$SystemReplyFunnel(6)*. This is used in conjunction with the *SystemReplyLink* by various intrinsics provided in the System 6400.
- *MS\$ParentLink(2)*. Points to your parent process. This link and the *MS\$ParentFunnel* allow your process and the parent process to communicate. This link has delete, copy, and pass notification rights.
- *MS\$NSMlink(4)*. Points to the Namespace Manager, which is the system process that manages files and directories. This is used for all file access.

- *MS\$SystemReplyLink(6)*. Points into the system reply funnel. This link is used by various system intrinsics when they need a have a brief conversation with another system process. The intrinsics send a copy of the system reply link with a request, and read the reply on the system reply funnel. The rule for using this link is that after sending a copy of it, your process should read all the expected replies on the system reply funnel before exiting or calling another routine.

UNIX Only

- *UTOK_LINK(2)*. User process to kernel link. This is used for making kernel requests (for example, system calls).
- *KTOU_FUNID(3)*. Kernel to user funnel ID. This is the destination for messages from the kernel to a user process (for example, replies to system calls).
- Additional links can be acquired with the assistance of the E-L Driver. (See *man el* and *man emboslogin*.)

SERVER RENDEZVOUS

A process can “connect,” or exchange links with another process in the same job by passing links through the intermediate processes in the job tree. (Under UNIX, the kernel provides this service. See *man el*.) To connect with a process that is not in the same job, such as one of the system server processes, requires a *server rendezvous*. A server rendezvous is an entry in the NameSpace, and therefore is identified by a pathname, just like a file or directory. However, instead of containing disk space, a server rendezvous contains a link to the process that created it (which is known as the “server”). Server rendezvous are not used for communication between UNIX processes, but they may be used to connect a UNIX process with an Embos, EMS, or System Foundation process.

A server rendezvous is opened by its pathname the same way an Embos file is opened: an Open request with a reply link is sent to the NameSpace Manager (NSM) via the NSM link. After checking security, the

Message System

NSM passes the Open request and the reply link to the server process via the link contained in the server rendezvous entry. The server process then validates the Open request, and sends an Open reply message with a link to be used for further requests. At this point, the requesting process and the server process both hold links to each other, and are connected. This connection is broken under these conditions: when the requesting process does a Close request, when it terminates, or when the process explicitly deletes the link.

Sometimes a prolonged connection to a server rendezvous is not required. Many system services, such as operator job control, require only a single request and reply. In these cases, additional data is included with the Open request, which is interpreted by the server process as a special request rather than a standard Open. Instead of returning an Open reply message with a link, the server process returns only a reply to the special request, then deletes the link to the requesting process. This method avoids the need for the requesting process to later send a Close request.

The most common use of server rendezvous is for accessing terminals, tapes, and other peripheral devices. An Embos or System Foundation device name such as */dev/term12* is really a server rendezvous created by the Access Manager for the device. Opening a device establishes a connection between the opening process and the access manager. A server process has the option of processing additional parts of the pathname appended to the server rendezvous name.

For instance, if the server rendezvous is */remote*, Open requests for pathnames such as */remote/a/b* are also forwarded to the server, as are create, delete, and other requests. This allows a server to implement an entire directory and file system. The access manager for labelled tapes uses this facility by creating a server rendezvous named */dev/tape*, so that pathnames such as

/dev/tape/tapeVolumeName/tapeFileName

can be used to open specific files on a labeled tape.

For information on creating server rendezvous and sending special requests to servers, see the Help files for FS\$CreateServerRzv and FS\$SendServerRequest respectively. File Open requests are sent to servers with FS\$Open, just as for normal files. These same routines are used under UNIX, employing the `elxsi_call` routine to correctly format the parameters. (For more information, see `man elxsi_call`.)

EXAMPLE STRATEGIES

Parent and Child Communication

Under UNIX, the *pipe* mechanism is usually used to communicate between parent and child processes. To set up direct message links, see `man el`. The rest of this discussion applies to Embos and EMS only.

The easiest way to communicate between a parent and child process is to use the links that are automatically set up when the child process is spawned. If your application is a parallel-processing program that forks multiple copies of itself, see Chapter 7, "Parallel Processing." If the parent process spawns a separate child process, you should use PM\$Spawn (for information on this, see the Help file). After a successful spawn, you can use PM\$GetLinkInPD and PM\$GetFunnelInPD to get the link ID and funnel ID to the new child.

If ParentCanHandleStatStack is True, the child process sends messages containing its termination status (via its parentLink) when it terminates. If this termination is unexpected, it is difficult for the parent process to identify those messages. Therefore, we recommend that you either call PM\$SetParentCanHandleStatStack(pd, false) before the PM\$Spawn, or have the parent send the child a new link into a different funnel when the communication begins.

Most programs (such as the Shell) take a deleteLink notification from the child as evidence of the child's termination. This is usually a reliable sign, but it may not be in the case of a destructive or damaged child process, which may simply delete its parentLink and continue running. (In fact, detached processes work in just this way.)

Message System

Structuring a General Server Process

A server process usually accepts requests from one or more sources, performs some work (which may involve sending and receiving other messages), and then sends a reply to the request. The request may contain the link on which the reply is to be sent, or the reply link may have been established earlier. The server will be easier to write if it need not be multi-threaded, that is, if it can process and reply to each request before accepting a new request. This is possible if the maximum delays in processing one request do not cause unacceptable delays in processing waiting requests. A server that handles many complex transactions may require multi-threading.

In any server process, new requests should be received with an `MS$ReceiveLink` or `MS$ReceiveLinkOnChannels` into an 888 byte buffer (or one or more buffers with a combined size greater than or equal to 888 bytes). (The UNIX intrinsics are `e_rcv_link` and `e_rcv_link_on_chans` respectively.) This is important even if requests are not expected to contain links or be that long, since a message that is too long or contains a link when none are expected is not removed from the funnel after a partially successful Receive attempt.

Most servers receive their requests via a server rendezvous. For information on `FS$CreateServerRzv`, see the Help file or the *Embos Programmer's Reference Manual* (Vol. 2). Under UNIX, servers can be accessed with the E-L Driver (see `man el`).

Single-Threaded Servers

If a server accepts only simple transactions, such as a request from a user process that requires only a single reply, it is usually best to require that each request contain a link for the reply. Then the server can simply receive the request, process it, send the reply, and delete the reply link all in one operation. If the server's transactions are more complex and involve multiple requests and replies (such as Reads of an open file), the reply link should be established when the server rendezvous is first opened, and reused as additional requests are pro-

cessed. A separate request link with a unique link code should be passed to each user process, so the link code can be used to identify the corresponding reply link, as well as any state information that is saved between requests. Often the link code is an index to a control block that contains the reply link ID and state information.

Some server processes accept several types of requests, which may have incompatible message formats. For example, the data in a file position request may not be distinguishable from a timer interrupt message. A common solution to this problem is to receive each type of request on a separate funnel, using the funnel ID to identify the type of request. One common use of multiple funnels is to separate Open messages from actual request messages. Using a separate funnel for each type of request also allows priorities to be assigned to request types by moving the funnel to a channel with the appropriate priority. Since the total number of funnels is limited to 255, and because of the problems associated with dynamically deleting funnels (see "Creating and Deleting Funnels" earlier in this chapter), creating a separate funnel per requesting process is discouraged.

A third reason to move funnels to different channels is to avoid conflicts with system intrinsics, which use Channel 15 to receive message replies. A call to `MS$ReceiveLinkOnChannels` with all channels enabled could result in receipt of a message intended for a system intrinsic. This problem can be avoided by moving the request funnels to Channels 2 to 14 and enabling only those channels in the channel mask. (Channel 1 is used for exception processing.)

Under UNIX, kernel replies are delivered to Channel 3. Channel 15 is not used unless Embos/System Foundation intrinsics are called.

Message System

To simplify decoding request and reply messages, it is useful to have a few common fields in fixed locations at the start of the message. Request messages typically begin with a request code that identifies the type of request, and a request tag. Reply messages typically begin with a reply code that identifies the type of reply, the request tag, and a status value. The server copies the request tag from the request to the reply, allowing the requestor to match the reply with the original request. The actual formats used by most System Foundation processes are shown in Table 5-4:

REQUEST FORMAT		REPLY FORMAT	
Bytes	Content	Bytes	Content
1-2:	Request code	1-2:	Reply code
3-4:	Unused, set to zero	3-4:	Unused
5-8:	Request tag	5-8:	Request tag
9-?:	Request-specific data	9-12:	Status value
		13-? :	Reply-specific data

Table 5-4. The Message Formats

An example in pseudo-Pascal follows. Requests from new user processes are received on NewReqFun, which is attached to NewReqChan. The user processes are given a link into OldReqFun, on which they can make subsequent requests. Their initial (new) request contains the reply link for both the new request and all subsequent (old) requests. The reply links are stored in control blocks (CBs), which are allocated as needed. Pointers to CBs are stored in the CBarry; Find-FreeCBix finds a free (nil) entry in this array. A control block is deallocated when the user process deletes the link corresponding to that control block.

```
const
    newReqChan    = 14;
    oldReqChan    = 13;
type
    CBptrType      = ^CBtype;
    CBarrrayType   = array[1..maxCBs] of CBptrType;
    CBtype = record
        replyLink  : MS$LinkIDtype;
        {other useful stuff}
    end;
var
    CB          : CBtype;
    CBix        : integer;
    CBarrray : CBarrrayType := maxCBs of nil;
    createParam : MS$CreateLinkParamType;
    rcvBuf      : machine record
        rcvParam : MS$ReceiveOnChannelsParamType;
        rcvBuf   : array[1..MS$MaxMessageLength] of char;
    end;
    newReplyBuf : machine record
        passParam : MS$CopyPassLinkParamType;
        replyBuf   : newReplyBufType; {New requesters}
    end;
    oldReplyBuf : machine record
        sendParam : MS$SendParamType;
        replyBuf   : oldReplyBufType; {Old requesters}
    end;
begin
    rcvBuf.rcvParam.typeOfReceive := [MS$Synchronous];
    rcvBuf.rcvParam.channelMask  := [newReqChan, oldReqChan];
    while true do begin
        rcvBuf.rcvParam.preferredLinkID := 0;
        MS$ReceiveLinkOnChannels( adrord(rcvBuf),
            MS$MaxMessageLength, 0, 0 );
        case rcvBuf.rcvParam.controlInfo.funnelID of
            newReqFun: begin
                CBix := findFreeCBix;
                new( CB );
                CBarrray[CBix] := CB;
                CB^.replyLink := rcvBuf.rcvParam.preferredLinkID;
```

Message System

```
{ Make new request link to send to user }
with createParam, linkRights do begin
    funnelID :=
        oldReqFun;
    preferredLinkID := 0;
    linkCode := CBix;
    notificationRights :=
        [MS$InformOnDelete];
    end;
MS$CreateLink( adrord(createParam) );
newReplyBuf.passParam.linkID :=
    CB^.replyLink;
newReplyBuf.passParam.linkIDtoCopyPass :=
    createParam.preferredLinkID;
MS$PassLink( adrord(newReplyBuf),
    size(newReplyBufType) div 8, 0, 0 );
end;
oldReqFun: begin
CB:=CBarray[rcvBuf.rcvParam.controlInfo.linkCode];
if MS$LinkDeleted in
    rcvBuf.rcvParam.controlInfo.msgType then
begin
    MS$DeleteLink( CB^.replyLink );
    dispose( CB );

CBarray[rcvBuf.rcvParam.controlInfo.linkCode] :=
    nil;
    end
else begin {Process the request.}
    oldReplyBuf.sendParam.linkID :=
        CB^.replyLink;
    MS$Send( adrord(oldReplyBuf),
        size(oldReplyBufType) div 8, 0, 0 );
    end;
end;
end;
```

Multi-Threaded Servers

Multi-threaded servers are generally written with a single call to `MS$ReceiveLinkOnChannels` into a maximum size buffer (888 bytes). Messages are sorted in much the same way as in a single-threaded server, but there is the additional complication that a message may represent not only a request from a user process, but a reply to a request that the server has sent to another process. In this case, a request tag is especially valuable since it can hold a control block address.

If the server allows only one request in progress at a time per user process, the user control block can simply contain state information, and the request tags of requests the server sends elsewhere can contain the addresses of the user control blocks. When these replies arrive, the user control block is found and the state updated. If the server allows multiple pending requests per user process, it may be necessary to allocate a work-in-progress entry (WIP) for each new request that cannot be completed immediately. The WIP contains all the information necessary to continue processing the request when replies from other processes arrive. When this is necessary, the request tags in messages to other processes can contain the WIP addresses.

A multi-threaded server that must also perform file I/O should use `no-wait I/O` and the `FS$ProcessIO` intrinsics. These allow the server to receive a message, decide whether it is a file system message, and if so, pass it to `FS$ProcessIO` for processing. The easiest way to tell that a message should be handled by the file system is as follows:

```
FD := FS$ReturnFD( FunnelID);
if FD <> nil then
  begin
    status := FS$ProcessIO( FD, adrord(Message), tag);
```

where *Message* is the buffer that contains the message, and *FunnelID* is the funnel on which the message was received.

The UNIX kernel (which is a multi-threaded server) and KAM (the Keyed Access Manager) employ an alternate method of implementing a multi-threaded server. While the details of this method are too complex

Message System

to discuss here, the desired result is to maintain a separate stack for each possible requestor. The server's stack pointer is changed to the appropriate stack when it decides which request to process next.

A simple "Return" puts the server back into the state it was in when the server was interrupted while processing a request (probably waiting for an external event). Once the stack switching mechanism is in place, this avoids the need for `WorkInProgress` entries and can make programming a multi-threaded server a straightforward matter. For more information, UNIX source licensees can look at the *sleep*, *idle*, *wakeup*, and *swtch* routines in the UNIX kernel.

Requesting a Service From a Server

If the server accepts requests through a server rendezvous, use the `FS$-SendServerRequest` intrinsic (see the Help file for details). If the server sends a single reply that completes the request, you can use the `MS$-SystemReplyLink` as the reply link and read the reply on the `MS$-SystemReplyFunnel`. Beware of error conditions: do not exit without reading the reply; nor should you call system intrinsics (other than the `statstack` and `string` package) between the time you send the request and read the reply because the system intrinsics might use the system reply link. The following code is an example of how to do this:

```
{ Assuming TAM is some type of server, send TAM }
{ an Open request.}
{ It is up to TAM to decode and understand the }
{ SpecialData string.}

var receiveParms: MS$ReceiveParamType;
    replyBuf:      array [1..MS$MaxMessageLength]
                  of char;
    specialData:   string;
    ...
SpecialData := 'Param1=Xyzzz Format=8';
Status := FS$SendServerRequest('TAM',
                               FS$DefaultDataAccess,
                               FS$DefaultEntryAccess,
                               MS$SystemReplyLink, SpecialData);

{ Now receive the reply on the System Reply Funnel. }

receiveParms.funnelID := MS$SystemReplyFunnel;
receiveParms.typeOfReceive := MS$Synchronous;
MS$Receive( adrord(receiveParms), 0, adrord(replyBuf),
            MS$MaxMessageLength);
```

Writing an Interrupt Routine

The important parts of an interrupt routine are as follows:

- The arguments are declared as a pair of integers.
- Exactly one message on the funnel for which this is the interrupt handler is received or otherwise disposed of.
- The interrupt routine terminates with an `OS$Ixit` call (or an `IXIT` instruction) whose parameter is the second argument to the interrupt routine.

An interrupt routine should not simply branch back to mainline code; this requires delicate handling of the stack marker that has been pushed by the interrupt. An interrupt routine can reference global variables (common variables in FORTRAN or static variables in C) and Send and Receive messages on any links and funnels.

Message System

An interrupt routine written in Embos C must be compiled with the *+nonC* option. An interrupt routine written in FORTRAN must have its arguments declared *%val* and must call *OS\$Ixit* with its argument marked *%val*. To deal with exception conditions under UNIX, users normally write signal handlers. This is highly recommended. However, if you must write an interrupt handler, you should write it in Assembler. The following interrupt routine sets a global flag each time it is called and deletes the message (in lieu of receiving it):

```
procedure myInterruptRoutine(  
    r0, r1      : integer);  
begin  
    MS$DeleteMessage( intFun );  
    globalFlag   := true;  
    OS$Ixit( r1 );  
end;
```

USING PASCAL

To obtain the type and routine declarations described here, include the file */embos/include/ms\$include.p* when using the message system from Pascal. The only item unfamiliar to Pascal programmers is likely to be the *adrord()* function, which returns the address of its argument. Pascal provides built-in access to the most common message system instructions: Receive, ReceiveLink, ReceiveOnChannels, Send, ReceiveLink-OnChannels, CopyLink, and PassLink.

These instructions behave similarly to their *MS\$XXX* counterparts, with these exceptions: if the last two parameters to the *MS\$* routine would be zero they can be omitted, and instead of returning status on the statstack they return status either as a function result or as their last parameter. For example, *MS\$Send(adrord(a), b, 0, 0);*

is equivalent to:

```
status := send(adrord(a), b); or  
send(adrord(a), b, status);
```

USING FORTRAN

The message system is more difficult to use from FORTRAN because it makes substantial use of control blocks with packed fields. In FORTRAN, the easiest way to handle this is to allocate an array of BYTE the size you need and then EQUIVALENCE various INTEGER types at the correct locations in the program. Examine the Pascal record definitions to see where the various fields go. Pascal sets can be built by equating the set elements to integers and then summing them to build the set. The following code is an example for the ReceiveParamType:

```
c  MS$messageTypeType      =
c    set of (MS$notUsed1, MS$notUsed2, MS$notUsed3,
c              MS$smallMessage, MS$linkDeleted,
c              MS$linkCopied, MS$linkPassed,
c              MS$includesLink);
c  MS$receiveType          =
c    set of (MS$unUsed1, MS$unUsed2, MS$unUsed3,
c              MS$unUsed4, MS$unUsed5, MS$dismissOption,
c              MS$interrogate, MS$synchronous);
c
c  MS$receiveControlInfoType = machine record
c    linkCode      : MS$linkCodeType      at 0 use 16;
c    funnelID      : MS$funnelIDtype      at 16 use 8;
c    messageType   : MS$messageTypeType   at 24 use 8;
c    fromProcess   : MS$processIDtype      at 32 use 16;
c    numBytes      : 0..MS$messageLengthUpb at 48 use 16;
c    end;
c
c  MS$receiveParamType = machine record
c    funnelID      : MS$funnelIDtype      at 0 use 8;
c    typeOfReceive : MS$receiveType      at 16 use 8;
c    preferredLinkID : MS$linkIDtype      at 32 use 16;
c    controlInfo   : MS$receiveControlInfoType at 64 use 64;
c    end;
c
```

Message System

```
    PARAMETER (dismiss=4), (interrogate=2),  
    & (synchronous=1), (linkDeleted=8),  
    & (linkCopied=4), (linkPassed=2),  
    & (includesLink=1)  
    BYTE rcvParam(16), buf(888)  
    INTEGER *1 funnelID, typeOfReceive, messageType,  
    & rcvFunnel  
    INTEGER *2 preferredLinkID, linkCode, fromProcess,  
    & numBytes  
    EQUIVALENCE (rcvParam(1), funnelID),  
    & (rcvParam(3), typeOfReceive), (rcvParam(5),  
    & preferredLinkID), (rcvParam(9), linkCode),  
    & (rcvParam(11), rcvFunnel),  
    & (rcvParam(12), messageType), (rcvParam(13),  
    & fromProcess), (rcvParam(15), numBytes)  
  
    funnelID          = 6  
    typeOfReceive     = synchronous + dismiss  
    preferredLinkID   = 0  
    CALL MS$ReceiveLink( rcvParam, 0, buf, 888 )  
    IF (AND(messageType, includesLink) .NE. 0) THEN  
c Handle message with link  
        ELSE  
c Handle message without link  
        END IF
```

Message System

Here is an example of an interrupt routine in FORTRAN that sets a flag and returns.

```
subroutine intHandler(%val r0, %val r1)
integer r0, r1
common/intCom/intFunnel, intOccurred
integer intFunnel
logical intOccurred

call MS$DeleteMessage(intFunnel)
intOccurred = .true.
call OS$Ixit(%val r1)
end
```

Message System

USING EMBOS C

To use the message system from the C language under Embos, put this include statement at the beginning of your source file:

```
#include <ms$include.h>
```

This declares all the MS\$ routines and types needed by the message system. The message system is then almost as easy to use in C as it is in Pascal. Remember to include an ampersand (&) when passing a parameter declared as `var` in the routine's Help file. For example, the following program demonstrates interrupt handling in Embos C and prints "done" after returning from `IntHandler`:

```
/* Demonstrates interrupt handling in C. */
/* Must be compiled with +nonC option. */

#include <stdio.h>
#include <ms$include.h>

/* intFun is shared between Main, which sets up the */
/* interrupting funnel, and intHandler, which must */
/* receive or delete messages on the funnel.*/

MS$funnelIDtype intFun = 0;

/* A C routine to install an interrupt handler */
/* must reference the interrupt handler via an */
/* "elxsi" declaration no matter what language */
/* the interrupt handler is written in. */

main() {
  elxsi INTHANDLER();
  MS$channelIDtype      intChan = 2;
  MS$sendParamType      sendParam;
  MS$createLinkParamType createParam;
  MS$channelMaskType     chanMask = 0x2000;
  int                    intHandlerAddr = &INTHANDLER;
```


Message System

```
/* Create a funnel, make INTHANDLER the interrupt */
/* handler for it, attach it to Channel 2, and */
/* make Channel 2 interrupting. */
MS$CreateFunnel(&intFun);
MS$SetFunnelInterruptVector(intFun, &intHandlerAddr);
MS$AttachFunnel(intFun, &intChan);
MS$EnableChannelInterrupts(&chanMask);
/* Create a link into the funnel and send a */
/* message down the link. */
createParam.funnelID    = intFun;
createParam.linkID      = 0;
MS$CreateLink(&createParam);
sendParam.linkID = createParam.linkID;
MS$Send(&sendParam, 0, 0, 0);
/* Prints "done" after returning from intHandler */
printf("Done\n");
}
```

```
/* A C language interrupt handler must: */
/* 1. Be compiled with the +nonC option. */
/* 2. Declare two "int" arguments. */
/* 3. Receive or delete the message that */
/*    caused the interrupt. */
/* 4. Return by passing its second argument */
/*    to OS$Ixit. */

intHandler(r0,r1)
int r0, r1; {
    elxsi OS$IEXIT();
    MS$DeleteMessage(intFun);
    printf("In intHandler\n");
    OS$IEXIT(r1);
}
```

Message System

USING UNIX C

The following program, called "int.c," demonstrates interrupt handling in UNIX C.

```
/* Program int.c demonstrates interrupt handling in UNIX C.
 * To compile, enter: cc int.c vec.s -lelxi      (BSD)
 *                  cc int.c vec.s             (System V)
 *                  cc -O -W2,-e int.c vec.s    (either)
 */
#include <sys/types.h>
#ifdef elxsibsd
#include <machine/emsg.h>
#else
#include <sys/emsg.h>
#endif

#define INT_CHAN 10      /* Channel on which to receive
                        * interrupt */
#define CHAN_MASK (0x8000>>INT_CHAN) /* Construct mask */
unsigned char intFun;    /* Funnel ID shared between
                        * main and intHandler */
extern int realIntVector(); /* Assembler routine */

main()
{
    cl_param      createParam;
    sn_param      sendParam;

    /* Create new funnel with interrupt vector realintVector. */
    /* Attach to the desired channel & enable interrupts on */
    /* that channel. */
    intFun = 0; /* We don't care which funnel ID we get. */
    e crt_fun(&intFun);
    e set_fun_int_vec(intFun, realIntVector);
    e attach_fun(intFun, INT_CHAN);
    e enable_chan_int(CHAN_MASK);
}
```

Message System

```
/* Create a link into the new funnel. */
createParam.cl_funid = intFun;
createParam.cl_linkid = 0;          /* Don't care */
e_crt_link(&createParam);

/* Send a 0-length message along that link. */
sendParam.sn_linkid = createParam.cl_linkid;
e_send_msg(&sendParam, 0, 0, 0);

printf("Done.\n");
}

/* This is a simplified C-interrupt handler. It is called
 * from a small Assembler module (in vec.s). The passed-in
 * point to the saved interrupt stack frame, which looks
 * like this: */

*   r1 ->    long long regs[16];    (regs[0] = .r0, etc.)
*           long long psw;
*           short    localpri;
*           char      unused[6];
*           int       unused2;
*   r0 ->    int       pc;
*/
intHandler(r0, r1)
int *r0; long long *r1;
{
e_del_msg(intFun);
    printf("In intHandler. Return PC=0x%x, SP=0x%x,
           r5=0x%llx\n",
*r0, (long)r1[15], r1[5]);
}
```

Message System

To run the *int.c* program presented above, the following Assembler file, *vec.s*, is required:

```
;
; Interrupt vector routine (file: vec.s)
; Hardware calls with .r0 and .r1 pointing into the
; interrupt stack frame.
; NOTE: Debugger stack traceback does not work properly
; with this routine.
    .text
    .extdef realIntVector
    .data 0 {16},152+16{16}          ; This is to help adb(1) with the
                                      ; stack frame.
realIntVector:
    subi.64 16,.sp                ; Room for params, temp and
                                   ; return address.
    st.32   .r0,[sp]4              ; Low-order half of r0 is the
                                   ; first argument.
    st.32   .r1,[.sp]8             ; Low-order half of r1 is the
                                   ; second argument.
    call    intHandler             ; Call C interrupt handler.
    ixit    16                     ; Exit through interrupt
                                   ; stack frame.
```

This chapter has presented all of the major concepts of the System 6400 message system, including message system constraints and guidelines on interprocess communication. We did not, however, discuss all the features of the message system. For reference information, see the *System Architecture* manual and the Help files for the MS\$ intrinsics.

CHAPTER 6

I/O MANAGEMENT

INTRODUCTION

In the System 6400, concurrent processing concepts have been extended from the operating system to the I/O system. Independent processes communicate via messages. Other processes view the controller-code on the device controllers as a process. Data can be transferred through the message system or through memory shared between a device controller and the host software process.

The I/O architecture allows I/O processing to be performed in parallel. Concurrent processing translates into higher throughput. That is, when I/O is organized to run in parallel, a great deal of I/O can be performed in any given frame.

Each I/O sub-bus can have up to sixteen device controllers; four of those controllers can be executing I/O transfers in parallel. Since each of the controllers is a processor, much of the work of managing a device is handled by the controller. In addition to the four controllers that can be transferring data, the other controllers on a sub-bus can also be doing useful work, such as rotational position sensing, seek optimization, and so forth.

For every class of devices, there is a supervisor, one or more access managers and/or UNIX device drivers, and one or more device controllers. The device controllers on the System 6400 generally perform functions that on other systems are handled by device drivers. These include processing interrupts and detection and recovery from low-level errors. The host-based supervisor process handles events such as device configuration and serious error processing. The access managers

I/O Management

or device drivers present a standard interface to applications and stand between an application and the controller.

The structure of the I/O system makes it convenient to create a new access manager or driver for each new controller or access method. The specialized code in these customized access managers results in greater reliability and performance. Performance is also enhanced because the servers can spread across the available CPUs. The house-keeping functions specific to the System Foundation are in the supervisor, eliminating the need for access managers or UNIX kernels to communicate with each other.

The I/O system for each class of devices diverges somewhat from this model. The rest of this chapter focuses on I/O to realtime devices, including the devices attached to the Parallel Interface controller, the Ethernet controller, and the VME subsystem. Although much of the general information presented in this chapter is relevant to the entire I/O system, details of the terminal, disk, and tape/printer subsystems are not included in this discussion. The normal system facilities provided for these devices suffice for most realtime needs.

MAJOR ELEMENTS

The major elements of the I/O System are Input/Output Processors, device controllers, supervisors, access managers, UNIX kernels, and various System Foundation processes. Only those major elements pertinent to realtime programming are discussed below.

INPUT/OUTPUT PROCESSOR

The main task of the Input/Output Processor (IOP) is to facilitate I/O transfers. The IOP allocates transfer channels and manages the message system for I/O processes. It is the interface to the Gigabus and manages access to main memory from the device controllers. Two I/O sub-buses are connected to each IOP, each of which can support up to sixteen device controllers.

A *transfer channel* is the physical conduit that gives a device controller the ability to transfer data from the controller to memory. Allocating a transfer channel means to allocate one of the four transfer channels the IOP has to one of the sixteen device controllers. Since there are four transfer channels per I/O sub-bus, four controllers can be transferring data at any one time. The 8 Mbytes per second of I/O bandwidth is divided among the four transfer channels, so that no single controller can access the entire I/O bandwidth.

I/O requests are made by sending a message to the appropriate device controller via the IOP. The IOP forwards these requests to the device controller. A device controller requests the IOP to receive the I/O request message by sending a "mini-message" request to the IOP via the sub-bus. When the operation is complete, the IOP notifies the controller and confirms "completion status."

Using an I/O Page Map, a data structure that lists the physical page numbers involved in each data transfer, the IOP translates virtual-to-physical memory address for data transfers between memory and the device controllers.

I/O SUB-BUS

The I/O Sub-bus is a 16-bit wide, four-way interleaved synchronous bus. It has an aggregate data bandwidth of 8 Mbytes per second. A single IOP supports two I/O sub-buses, which are usually referred to as 0 and 1. There are commands that allow a controller to instruct the IOP to send a message, start a main memory transfer, and so forth. These commands generally manipulate registers on the "front end" (or sub-bus interface) of the controller. The I/O sub-bus is usually accessible only to those who program the device controllers.

I/O Management

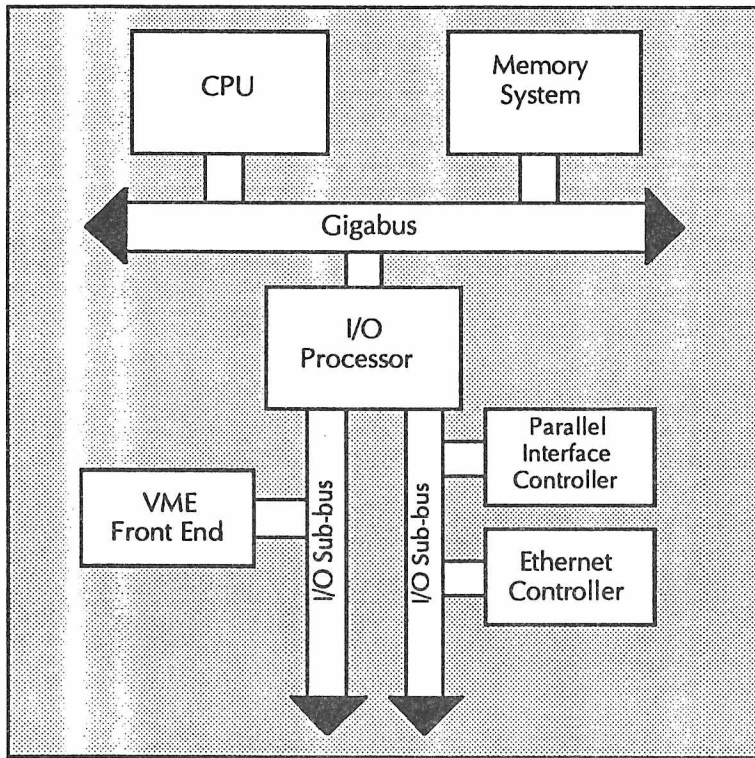


Figure 6-1. The Major Hardware Elements

PARALLEL INTERFACE CONTROLLER

The Parallel Interface Controller (PIC) is an integrated 68000-based microcomputer that emulates two DEC DR11 interfaces,¹ providing an industry standard parallel interface for many external devices. Specific applications on the System 6400 include array processors, plotter rasterizers, high-speed graphics, and computer-to-computer links. (Please refer to the appropriate Digital Equipment Corp. documentation

¹ DEC is a registered trademark of Digital Equipment Corporation.

I/O Management

for a full description of the hardware interface.) PIC hardware can be configured for either the DR11-W or DR11-B interfaces.

To provide the ability to manipulate the DR11 interfaces and manage the work queues on the device controller, the PIC is programmed in M68000 Assembly language. The PIC1 microprogram provides Read and Write commands, commands that sense the status of the interface, manage error handling on the controller, and so forth. (For a more detailed discussion, see the on-line document, *PIC1 Interface Specification*.)² You can reprogram the Parallel Interface Controller for application-specific functions.

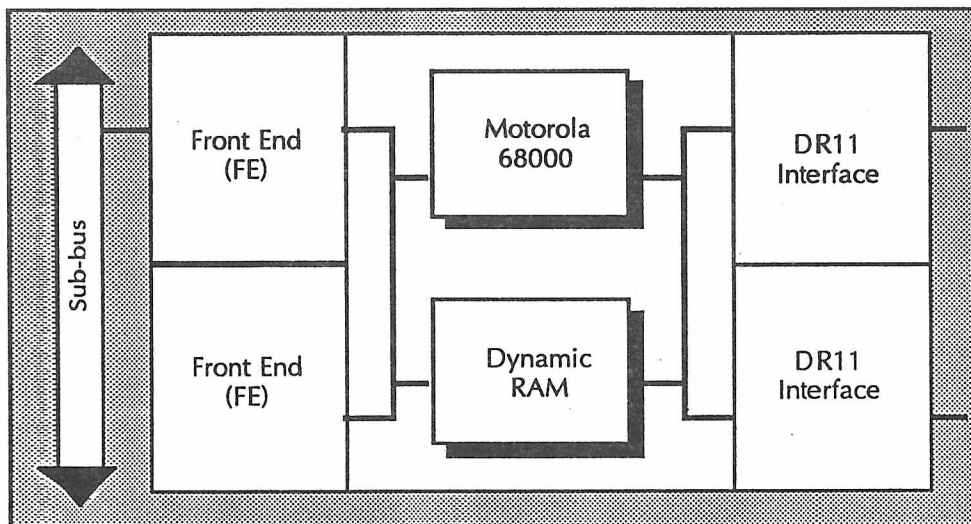


Figure 6-2. The Parallel Interface Controller

² Though the on-line documents can be read on-line, they are included on the system so you can print them. To print this document, enter the following command at the Embos command line:

```
printdoc Pic1Spec
```

I/O Management

THE ETHERNET CONTROLLER

The Ethernet Controller is an integrated 68000-based microcomputer that provides the physical interface to an Ethernet network. Please refer to the appropriate Xerox documentation for a full description of Ethernet. The controller's major components are the Dynamic Front End (DFE) sub-bus interface, the Motorola 68000 microprocessor, dynamic RAMs for buffering code and data, and the Local Area Network Controller for Ethernet (LANCE).³ These components are shown below in Figure 6-3.

The supplied controller program is written in M68000 Assembly language and the C language. The controller program supports sending and receiving ethernet packets, which are buffered in the controller's memory. The controller can be reprogrammed for application-specific functions.

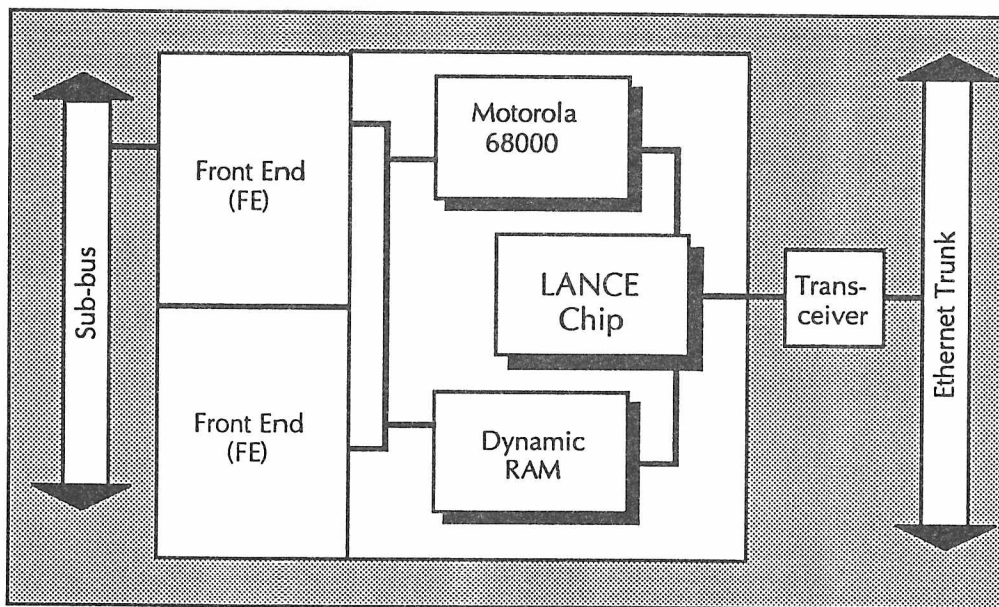


Figure 6-3. The Ethernet Controller

³ LANCE is a trademark of Advanced Micro Devices.

VME FRONT END

The VME Front End (VFE) connects the I/O sub-bus to an industry standard VME bus. It performs the functions of the Dynamic Front Ends (DFE) on standard System 6400 device controllers. The VFE is a double-high Eurocard Format module that is mounted in a rack in a separate VME chassis. The VFE is connected to the System 6400 with an external sub-bus.

HIGH-SPEED DEVICE

The High-Speed Device (HSD) is a general purpose 32-bit interface that serves as an HSD master and operates in either Normal or Interbus Link mode. Software on the controller implements all the driver functions. For a description of the software interface, see *The HSD Interface Specification*.⁴

The High-Speed Device consists of the VME Sub-bus Front End (VFE), a Motorola 68020 CPU card with memory, and one or more HSD Interface cards in a VME chassis. The HSD is connected to the System 6400 with an external sub-bus. These components are shown below in Figure 6-4.

4 To print this on-line document, enter
printdoc HsdSpec

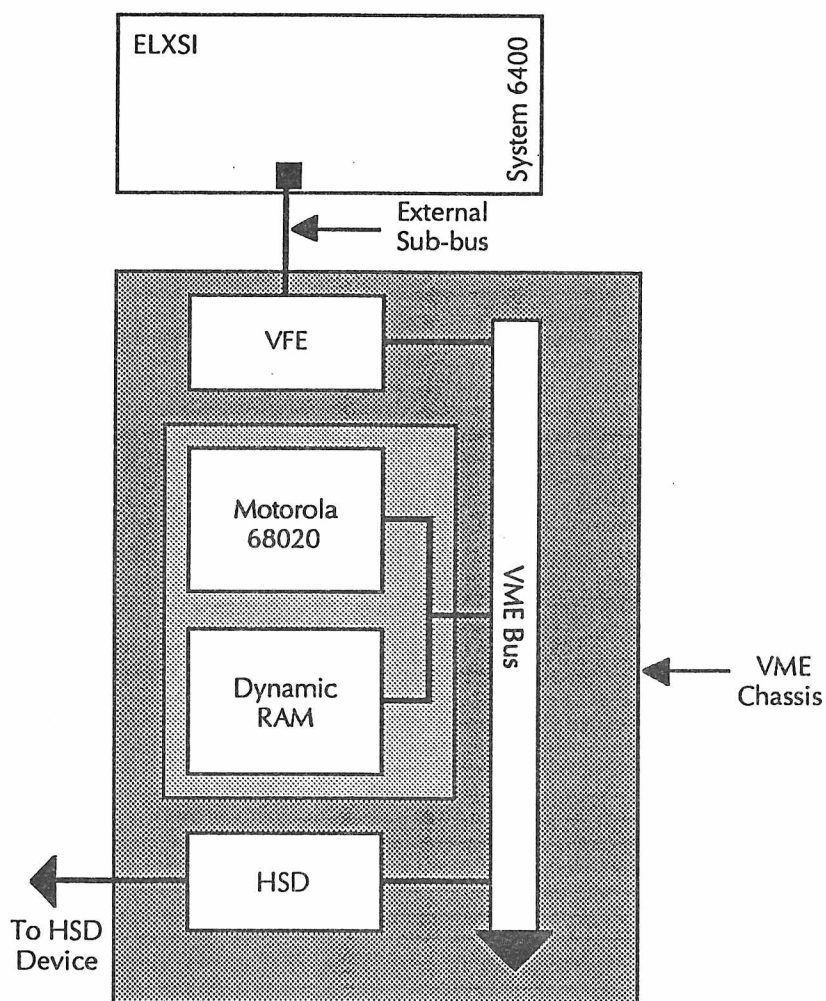


Figure 6-4. The High Speed Device Using the VME

CUSTOM CONTROLLERS

Custom controllers can be implemented with the VME Front End (VFE) sub-bus-to-VME-card.

COMMUNICATIONS SUPERVISOR

The main function of the Communications Supervisor is to manage controller configuration (including creating and deleting access managers), and downloading controller-code. The Communications Supervisor (CS) manages the Parallel Interface controllers, VME-based controllers, Ethernet controllers, and the other communication controllers on the system. The ELXSI Configurator program (Elcon) declares the controllers to the operating system and creates the initial set of links between the controllers and the CS. The CS is also the parent of all the access managers for these controllers.

The Communications Supervisor is part of the bootstrap image that is rolled in when the system is booted. Information about the controllers is passed to the CS; this information is part of the system image when the system is ELCONed.

THE MEMORY MANAGER

The Memory Manager is a system process that manages the virtual memory system. The System 6400 virtual memory is based on a demand paging mechanism with a page size of 2048 bytes. One of the services provided by the Memory Manager is the ability to perform I/O directly from a virtual memory address.

To perform I/O, a message is sent to the Memory Manager, specifying the virtual page numbers and the device controller process ID. The Memory Manager freezes the corresponding pages in real memory and creates the appropriate I/O page map entries. The software process receives an index to the page map entries from the Memory Manager; this index is used as the Physical-Virtual Address (PVA) for subsequent data transfer. (While this Memory Manager service can be used directly

I/O Management

by user programs, it is safer to have an access manager freeze memory for the user process.)

The PVA is passed via a message to the device controller. Then the controller and the IOP use the PVA to locate the regions of physical memory to be used for the data transfer. Data transfer takes place between memory and the controller via the IOP. (For more information, see Chapter 4, "Memory Management.")

REALTIME ACCESS MANAGER

The Realtime Access Manager (RTAM) gives applications written by customers direct access to device controller processes. RTAM allocates and deallocates devices attached to device controllers. When a device is allocated, RTAM passes the controller-links to the realtime application. The application can then communicate directly with the controller for that device. To allow a Realtime programmer to set up and "unset up" memory for I/O, RTAM also provides access to the Memory Manager. RTAM was designed to provide an example for those who need to develop custom access managers. Documented source code for RTAM is available for purchase. For more information, including programming examples, see "The Realtime Access Manager" later in this chapter.

PARALLEL INTERFACE ACCESS MANAGER

The Parallel Interface Access Manager (PIAM) is a block I/O access manager for devices attached to the Parallel Interface Controller. PIAM supports the Numerix array processor, Logic Science's HSR-11 rasterizer/plotter and a variety of computer-to-computer communications devices.

The PIAM is a generic access manager for PIC devices. Among other features, it supports "device classes," a concept that allows similar devices to be treated as a device pool. Through the **ConfigPICdevice** command, PIAM can be associated with a specific PIC port. PIAM manages all requests directed to a device; that is, the application's connection is actually to PIAM, rather than the PIC. PIAM communi-

cates with the Memory Manager to freeze or unfreeze user memory as necessary. (For more information on PIAM, refer to "The Parallel Interface Access Manager" later in this chapter.)

NAMESPACE MANAGER

The NameSpace Manager (NSM) enforces Embos, EMS, and System Foundation security and controls access to named objects in the system, such as disk files, devices, or other processes. There are several types of entries in the NameSpace, such as directories, disk files, aliases, rendezvous, and server rendezvous.

The server rendezvous allows an application to gain access to a server. After validating the security of an application's request, the NameSpace Manager forwards the request through the server rendezvous to the server.

File security is provided for every node in the NameSpace (except the alias nodes) through access control lists and program lists. An access control list is a list of users and groups that can access an object, along with the types of access allowed to each user and group.⁵ A program list disallows access to a file except by the programs specified in the list.

⁵ The categories of access are data access, entry access, and directory access. The types of access under these categories include read, write, delete, append, execute, and so on. For more information, see the Help file for the **SetSecurity** command.

I/O Management

DOWNLOADING FROM THE SVP

The file */local/hardware* on the SVP describes not only the functional units but the device controllers as well. A device controller is identified by its name, type, location, and revision level. The location of a controller is identified by the name of its IOP, the IOP sub-bus (0 or 1), and the sub-bus address. The version of the controller-code to be downloaded, which is based on the controller type and revision level, is identified in the file */mcode/pc.matrix*.

The **PCIML** command on the SVP uses these two files to download code into a controller. Controller-code can also be downloaded to some controllers from the supervisor, rather than from the SVP.

Before code is downloaded to a controller, the code is compiled or assembled for the specific processor used on that controller (for example, the 68000). Then the code is run through a post-processor, which formats the code so that PCIML can download the code to the controller. The post-processor 1) puts control information around the code that indicates where in the controller's memory the code is to be placed, 2) zeroes out memory, and 3) issues commands that allow the controller to start execution. The result is a file that can reside on either the SVP disk or the System disk.

The PCIML program reads the post-processed file and sends To-hardware messages to the I/O Processor (for the controller). To place code into the controller, PCIML 1) reads blocks of code from the post-processed PCIML file and 2) writes it to the device controller's memory (by sending To-hardware messages). Optionally, after the code has been written to the controller's memory, the PCIML program can send additional To-hardware messages that tell the controller to begin executing.

To-hardware messages are received and interpreted directly by the CPU or IOP, rather than the *process* on the CPU or the *controller* on the IOP. For example, when an application does a To-hardware Send to the IOP for a particular device controller, the IOP actually sees and receives the message and takes action based on the To-hardware message. Whereas

if you send a regular message to a controller process, the IOP is only a conduit, passing the message along, but having no knowledge of its contents.

WARNING
Errors by processes with the rights to send To-hardware messages can seriously compromise the integrity of the system.

CONFIGURING DEVICES

Site-specific device configuration information is kept in configuration Shellfiles (in the Embos */local/commands* directory) that run during start-up. This information consists of the names of all the devices, the names of the controllers that the devices reside on, the name of the access manager that manages the device, and other device specific information such as the terminal ID, Baud rate, and so on.

For each device, the configuration Shellfile sends the configuration information to the supervisor. The supervisor notes which access manager is going to manage the device, spawns the access manager if necessary, and forwards the configuration information to the access manager. The first time a particular access manager is assigned to a device, the supervisor spawns the access manager. Since one access manager can manage more than one device, the second time that access manager is assigned to a device, the access manager process does not have to be spawned--the configuration information is forwarded directly.

The access manager establishes communication with the controller, creates a rendezvous for access by the application, and replies to the device configuration command, either confirming that the device configuration is complete or indicating that a problem has been encountered.

I/O Management

Unlike a software process, a controller cannot create self-links, so all the links that point to a controller have to be created when the System is ELCONed. Generally these links have predetermined functions. Some of the links manage the device controller (the supervisor usually keeps these links) and other links are for particular devices on the controller, which the supervisor gives to the access manager. Supervisors do not usually access the devices.

The supervisor retains some state information about the devices, such as which access manager is managing which device, where the access managers are and their current state, and so on. During the system boot, the supervisor forwards (that is, copies) the controller-links to the appropriate access managers.

A supervisor can manage multiple devices (for example, the ATCS can manage hundreds of terminals). When the first device is configured, the supervisor passes the links for the first device to the access manager. As additional devices are configured, the supervisor passes the links for each device to the access manager (which may be a different access manager than the one assigned to the first device). At this point, the supervisor's job is done. The access managers are now responsible for managing the devices.

During a UNIX boot, the UNIX device drivers may rendezvous with access managers or supervisors to acquire direct links to controllers.

RESTRICTING ACCESS TO A DEVICE

Configuring a device for restricted access to a controller is a two-step process: 1) configure the device and 2) set the security such that access is limited to a user or group of users. All security is managed by the NameSpace Manager. When a Namespace entry is created, you can specify certain access rights such as passwords, user lists, or groups that determine who may have access to that entry.

After a device is configured, the system administrator can issue a **Set-Security** command that restricts access to the device by specifying the rendezvous for that device. When a device is opened, the NameSpace

I/O Management

Manager checks the security status of the device. If the NSM forwards the Open request to the server, the application has the rights to Open that device. The rendezvous for devices are located in the Embos */dev* directory (for example, */dev/term21*).

Configuration Example

The command to configure a realtime device is **ConfigRTDevice**. Its format is shown below:

```
ConfigRTDevice [device name] [controller name] [address of the  
device on the controller] [access manager name]
```

Let's assume your realtime device is named */dev/hsd1* and that the controller name is *vfe1*. The virtual address is *0* and the access manager is *RTAM*. Given this, you would enter the command as follows:

```
ConfigRTDevice device=/dev/hsd1 controller=vfe1 &  
address=0 AM=rtam
```

RTAM creates a server rendezvous named */dev/hsd1*. Applications can then use this rendezvous to gain access to that device through RTAM.

The next step is to limit access to a defined group or list of users. Assuming that you want to limit access to two users, Olson and Wilkes, this is the **SetSecurity** command that you would issue:

```
SetSecurity /dev/hsd1 olson,wilkes All
```

GAINING ACCESS TO A DEVICE

Once a device is configured, the access managers have a two-way communication path to the controllers and one or more rendezvous entries in the NameSpace that allow applications to gain access to a device. Opening a device rendezvous gains access to the access manager for that device. The File System Intrinsic FS\$Open performs all the steps necessary to open a device. When a device is open, the application can send requests directly to the access manager.

ACCESS THROUGH AN ACCESS MANAGER

There are two levels of access to a device: access through the access manager and direct access to the controller. How an application establishes a connection with the access manager is described below. (These activities are usually hidden in intrinsics such as FS\$Open, rather than being explicitly programmed.)

1. Each process is spawned with a link to the NameSpace Manager (NSM). The NSM has a link to an access manager for each server rendezvous. (Under UNIX, the NSM link must be explicitly requested. See man-page **emboslogin** (3).)
2. The application creates a self-link and passes it to the NSM, along with a request to open a NameSpace Entry.
3. The NSM searches its NameSpace file for the entry. If the entry is a server rendezvous and the request passes the security check, the NSM forwards the Open request (including the application's link) to the creator of the server rendezvous. In this case, the creator of the server rendezvous is an access manager.

If the Open request is rejected (which would usually be for security reasons), the NSM replies to the application with a "bad status" message.

4. The access manager creates a self-link and passes it to the application process. When the application process receives the link from the access manager, the device is open.

5. When FS\$Open is finished, the application has a communication path to the access manager.

For many devices, such as terminals and tapes, the application always interacts with a device through an access manager. For example, an application can issue an FS\$Read, which sends a message to the access manager; the access manager then performs the Read and replies to the application.

DIRECT ACCESS TO A CONTROLLER

When an application requires direct access to a device controller, the application must still open the device through the access manager. The application makes a request to the access manager, requesting the link to the controller; the access manager copies one of its controller links to the application. This puts the application in the same relation to the controller that the access manager usually holds: the application becomes responsible for sending messages directly to the controller.

Now the application has its own link to the controller, but there is no reply path. The application creates a self-link and passes it to the controller. This completes the two-way communication between the application and the controller.

Though the access manager is shunted to the side in this case, other devices on that controller can be managed in the usual way by the access manager, providing shared access to the other devices while the one device is reserved for special access by the application.

CLEANING UP

A process typically gives up access to a device by simply closing the device (with the FS\$Close intrinsic). FS\$Close sends a message to the access manager indicating that the process is finished talking to the device. The access manager may first need to communicate with the controller to terminate or wait for requests that may be in progress before it deallocates the device. The access manager then sends a reply

I/O Management

message to the user process, confirming the Close operation. At this point, the user process no longer has access to the device.

If a user process aborts, the access manager is notified of the abort by a DeleteLink notification; the access manager then proceeds to close access to the device as described above, except that the access manager does not attempt to send a reply message to the aborted process.

TRANSFERRING DATA

There are two ways to move data between a controller and an application--moving data directly from main memory to a controller's memory (or vice versa) via Direct Memory Access or passing data via the message system. Both methods of moving data are available, regardless of whether one is accessing a controller directly or accessing a controller via an access manager.

A process on the System 6400 makes a request for I/O to the controller by sending the controller a message. The CPU process that makes this request for I/O resumes its work as soon as the message is accepted by the IOP. The data transfer is completed without any further CPU involvement.

The controller is notified that there is a message pending. The controller asks the IOP to send the message to the controller's memory. The IOP moves the data in the message from main memory to the controller's memory. At this point, the controller has the message that contains the I/O request, but not necessarily the data itself.

A WRITE OPERATION

The following sequence describes a Write operation: If the data to be transferred is in the message, the data is moved to the controller's memory along with the request. But when the request does not contain the data, it contains instead 1) an address that indicates either where the data currently resides in main memory, and 2) the amount of data to be moved.

I/O Management

The controller asks the IOP to move the data from main memory (using the address it received) to the controller's memory; the IOP acts on that request. Once the data (or a portion of the data) is in the controller's memory, the controller starts the I/O to the device. This continues until all the data for that request has been transferred. The controller sends a reply message indicating a successful operation.

A READ OPERATION

This sequence describes a Read operation: If the data is to be sent in the message, the data will be sent in the reply once the operation is complete. If the data is not to be sent in the message, the Read request contains the address and the length of the buffer in main memory where the data is to be transferred. The controller initiates the input from the device to the controller's memory. Once the data (or a portion of the data) is in the controller's memory, the controller asks the IOP to move the data from the controller's memory to main memory. The controller sends a reply message indicating a successful operation.

The IOP does not control when I/O is performed; it only acts on the requests from the controllers to transfer data. When a controller turns on the bit that indicates it needs to transfer data, the IOP performs the transfer when a transfer channel is available. (The availability of transfer channels is rarely a problem.) Up to four controllers per I/O sub-bus can transfer data at the same time. The I/O sub-bus address prioritizes the requests: address 0 has the highest priority; address 15 has the lowest priority.

MEMORY ORGANIZATION

Software processes reference the memory allocated to them in virtual addresses. That is, the beginning of their memory space always begins at location 0, and the addresses are contiguous throughout their memory space. The actual physical placement of a process's memory, however, may be scattered across system memory or not even be resident in physical memory.

I/O Management

The CPU converts virtual addresses to physical addresses. (Page maps support this conversion.) Each software process has a page map: for each virtual page (2048 bytes) of memory, there is a corresponding entry in the process's page map called the *Page Map Entry* (PME).

A page map entry defines whether a page can be read, written, or executed. The PME also indicates where each page is located. If a page is physically in memory, the physical page address is also in the PME. When a page is resident in memory, the page map entry also contains an index into the Page Frame Table, which has an entry for each physical page in memory.

The I/O Page Map

The I/O Page Map (IOPM) is the data structure the IOP uses to 1) obtain the frame address of a particular page in main memory and 2) determine whether a device controller has rights to access the page. Like the Page Frame Table, the I/O Page Map has an entry for each page of physical memory.

Each I/O Page Map Entry (IOPME) corresponds to a single page of physical memory and contains 1) a frame address, 2) up to three controller IDs, and 3) the corresponding access rights for those controllers. An IOPME is referenced by a Physical-Virtual Address (PVA).

An entry in the I/O Page Map may be any of three levels. A level-1 entry indicates that the frame address is the page to be accessed by the transfer operation. A level-2 entry indicates that the frame address is for a page that contains a list of indices into the I/O Page Map (that is, a list of Physical-Virtual Addresses). A level-3 entry is for a page that contains a list of PVAs for level-2 entries.

SETTING UP FOR I/O

To ensure that a software process has its memory available for data transfer with a controller, it must request that the Memory Manager set up the data transfer. The maximum amount of data that can be transferred in one operation is 16 Mbytes. The software process gives the Memory Manager 1) the virtual address of the pages that are to be accessed and 2) the controller-process ID of the controller that will transfer the data. The Memory Manager accesses the page map entries for the requested virtual pages and checks to see if they are in memory.

If pages are not resident in physical memory, the Memory Manager invokes the disk to bring the requested pages into physical memory. To prevent pages being swapped back to disk while the peripheral device is transferring the data, the Memory Manager freezes the pages in memory.

If the software process requests a transfer of data contained on a single page, the I/O page map entry is a level-1 entry. For a single-page transfer, the Memory Manager places 1) the frame address for the page, 2) the controller-process ID, and 3) the access rights of the controller to that page in the I/O page map entry.

If the transfer is across several pages of memory, the Memory Manager first constructs an entry in the I/O page map for each page of data being transferred and then builds a list of Physical-Virtual Addresses. Since a list of 512 PVAs can fit on one page, a data transfer of up to 512 pages (1 Mbyte) can be represented by a level-2 entry. Requests for transferring more than 1 Mbyte require a level-3 entry.

TRANSFERRING THE DATA

When a controller transfers data, it provides the IOP with 1) the Physical-Virtual Address, and 2) an offset from the beginning of the page pointed to by the PVA. The IOP uses the I/O page map to locate the requested pages in physical memory and determine whether the controller can access the pages. If the device controller has access

I/O Management

rights for the page, the IOP moves the data. The controller sends a reply message to the application when the the transfer is complete.

THE REALTIME ACCESS MANAGER

The Realtime Access Manager (RTAM) provides direct access to most types of device controllers, including Parallel Interface controllers, Ethernet controllers, and VME-based controllers. Realtime applications have complete control of the interface to the device controller. RTAM provides some operating system services, but it does not interface directly with the controller.

CONFIGURING RTAM

When a device is configured, the system administrator selects RTAM as the access manager for that device. RTAM is spawned by the Communications Supervisor (if it is not already running), and receives the links to the controller for that device. It retains these links, but does not communicate directly with the controller. RTAM creates a server rendezvous for the device in the Namespace.

The following is an example of a device configuration command:

```
configRTdevice realPIC controller=pic1 &  
address=10 type=user am=RTAM
```

ACCESS VIA FILE SYSTEM INTRINSICS

A device managed by RTAM can be opened with the FS\$Open intrinsic. RTAM exchanges links with the realtime application. Note that after the Open, the application has a link to RTAM, not the controller for the device. When the device is opened, it is allocated to the Realtime application. When the device is closed, the device is deallocated and all pages set up for I/O are released.

RTAM supports FS\$IOCTL requests to return configuration information, set up pages for I/O, and return links to the controller. None of these requests require RTAM to communicate directly with the controller.

RTAM does not support any of the data transfer requests (such as FS\$Write, FS\$Read, and so on).

FS\$IOCTL performs a control operation on a device. The request codes are defined by the access manager for the device.

```
function FS$IOCTL
(  FD                :  FS$FileDescType;
  Request            :  integer;
  BufferAddress       :  $AddressType;
  Size               :  integer )
:  $statusType;
```

Parameters

FD. A file descriptor for an open device. (Input)

Request. Device-dependent request code. (Input)

BufferAddress. The usage of this parameter is determined by the request and the device access manager. Usually, it is the address of the buffer involved in the IOCTL operation.

Size. The usage of this parameter is determined by the request and the device access manager. Usually, it is the size of the buffer involved in the IOCTL operation.

Table 6-1 shows the request codes supported by RTAM.

I/O Management

Name/Hex Value	Function
FS\$SetupIOBufferIOCTL #0000 0008	Sets up buffer for I/O. The PVA and offset are returned in ReturnParm1 and ReturnParm2. (See the Help file for FS\$GetReturnParms.)
FS\$ReleaseIObufferIOCTL #0000 000C	The buffer that was set up for I/O is un-set up. The BufferAddress and Size must be identical to the values used for set up.
FS\$DevInquireIOCTL #0000 000D	Returns device information in the ReturnParms. The BufferAddress and Size parameters are not used and should be set to zero. (See Table 6-2 below.)
FS\$ForwardLinkIOCTL #0004 000E	Send a link to the device controller. The access manager may have more than one link to the controller, numbered 0 to n. The Size parameter identifies which link to return. The BufferAddress parameter is not used and should be set to zero. If this request code is successful, the application must call FS\$ReturnIOCTLLinkID to obtain the linkID of the new link.

Table 6-1. RTAM Request Codes

	0	16	31
ReturnParm1	Device type		Device subtype
ReturnParm2	Zero		Device number
ReturnParm3	Zero		Controller ID
ReturnParm4	Zero		Number of links
ReturnParm5	Opens		Unique number
ReturnParm6	Unused (zero)		
ReturnParm7	Unused (zero)		
ReturnParm8	Unused (zero)		

Table 6-2. Return Parameters

FORTRAN EXAMPLE FOR RTAM

This sample Fortran program, called RTAMUSER, 1) opens the device, 2) retrieves some configuration information, 3) sets up the buffer for I/O, and 4) gets a direct link to the controller. The program does not make controller requests, but there are comments that indicate where these requests would be placed.

```

PROGRAM RTAMUSER
  IMPLICIT NONE
  integer*4 FS$ReadWriteDataAccess
  Parameter(FS$ReadWriteDataAccess      = 3)
  integer*4 FS$SetupIObufferIOCTL
  Parameter(FS$SetupIObufferIOCTL      = 16#00000008)
  integer*4 FS$ForwardLinkIOCTL
  Parameter(FS$ForwardLinkIOCTL      = 16#0004000E)
  integer*4 FS$DEVInquireIOCTL

```

I/O Management

```
Parameter(FS$DEVInquireIOCTL      = 16#0000000D)
integer*4 MAXBUF
Parameter(MAXBUF = 65536)
INTEGER*4 IOBUFF(MAXBUF)      !The I/O buffer
INTEGER*4 CONTID,DEVNUM      !Controller ID and device no.
INTEGER*4 DUMMY
INTEGER*4 PVA, PVAOFF      !PVA and offset
INTEGER*4 LINK0, LINK1      !Links to the controller
INTEGER*4 FD      !File descriptor for the
                        device
INTEGER*4 KIND      !Device kind and unused
INTEGER*2 KINDS(2)
EQUIVALENCE (KIND, KINDS(1))
PRINT *, '*** RTAM User Program ***'

C
C  OPEN the Device      pathname=/dev/readD
C
    if (FS$OpenOld(FD, '/dev/readD', FS$ReadWriteDataAccess)
*      .NE. 0) goto 995
    if (FS$Open(FD) .NE. 0) goto 995

C
C  Get the controller ID and device number.
C
    if (FS$IOCTL(FD, FS$DEVInquireIOCTL,
*      0, 0) .NE. 0) goto 995
    CALL FS$GetBlockIOreturnParms(  FD,
*                                  KIND,
*                                  DEVNUM,
*                                  CONTID,
*                                  DUMMY, DUMMY, DUMMY, DUMMY, DUMMY)
    PRINT *, 'KIND=', KINDS(1), ' SUBKIND=', KINDS(2),
*      ' CONTROLLER PID=', CONTID, ' DEVICE=', DEVNUM)

C
C  Set up the I/O buffer.
C
    if (FS$IOCTL(FD, FS$SetupIObufferIOCTL,
*      IOBUFF, MAXBUF)
*      .NE. 0) goto 995
    CALL FS$GetBlockIOreturnParms(FD, PVA, PVAOFF,
*      DUMMY, DUMMY, DUMMY, DUMMY, DUMMY, DUMMY)
```

```
C
C   Get links to the controller.
C
    if (FS$IOCTL(FD, FS$ForwardLinkIOCTL,
*       0, 0) .NE. 0)
*   goto 995
    LINK0 = FS$ReturnIOCTLlinkID(FD)
    if (FS$IOCTL(FD, FS$ForwardLinkIOCTL,
*       0, 1) .NE. 0)
*   goto 995
    LINK1 = FS$ReturnIOCTLlinkID(FD)
C
C   Now you can access the controller directly.
C   Insert the controller-specific code here.
C   Close the device.
C
    if (FS$Close(FD,0) .NE. 0) goto 995

    GOTO 999
995 CALL $ERRORMSG()
    STOP 'ERROR REPORTED FROM RTAM'
996 STOP 'ERROR REPORTED FROM CONTROLLER'
999 STOP 'SUCCESSFUL COMPLETION OF TEST'
1000 END
```

I/O Management

THE PARALLEL INTERFACE ACCESS MANAGER

The Parallel Interface Access Manager (PIAM) is a general-purpose access manager for the parallel interface controller that provides the functionality of a DR11 interface. When used in conjunction with PIAM, the controller can be used as an interface to a user device or as a link between two systems. For a complete description of these functions, consult the appropriate DEC manuals.

CONFIGURING PIAM

When a device is configured with the **ConfigPICdevice** command, the system administrator selects PIAM as the access manager for that device. The command then goes through a server rendezvous to the Communications Supervisor.

The CS spawns PIAM (if it is not already running); if PIAM is already running, it is not spawned again. (A single PIAM can support any number of devices.) The CS then forwards the device configuration request to PIAM, along with links to the controller. PIAM creates a server rendezvous for the device in the NameSpace, sends the controller a reply link, completing two-way communication between the controller and PIAM, and exchanges configuration messages with the device controller. When the configuration is complete, the **ConfigPICdevice** command receives a reply message confirming the completion status.

The ConfigPICdevice Command

The **ConfigPICdevice** command configures a device that is attached to the System 6400 through the Parallel Interface Controller (PIC). The physical device is identified by its controller name and a controller-relative address. This command requires system operator capability.

The **ConfigPICdevice** command has four required parameters: *device*, *controller*, *address*, and *type*, and several optional parameters, including the name of the access manager. (See the Help file on the **ConfigPICDevice** command for information on all the parameters.)

I/O Management

device The pathname of the namespace entry to be created for the device. The default directory for devices is */dev*.

controller The name of the device controller to which the device is attached. The name should be identical to the name in the hardware file used to Elcon the system.

address The controller-relative address of the device. The values for this parameter may be 10 or 11.

type The type of device.

AM The name of the access manager bound file in the AMPic directory. The default is "PIAM."

Two examples of the **ConfigPICdevice** command are presented below. In the first example, the required parameters are explicitly stated. In the second example, the four required parameters are defined by position (device=ARA1, controller=PIC1, and so on).

```
configPICdevice device=remotel controller=PIC1 &  
address=11 type=ELXSlink
```

```
configPICdevice ARA1 PIC1 10 MARS432
```

ACCESS VIA FILE SYSTEM INTRINSICS

Most access to PIAM is through the File System Intrinsics, which manage the link to the access manager and the funnel for the reply link. Intrinsics exist for both synchronous and asynchronous requests.

- **FS\$Write** (or **FS\$InitiateWrite**) transfers data from memory to the device.
- **FS\$Read** (or **FS\$InitiateRead**) transfers data from the device to memory.
- **FS\$IOCTL** (or **FS\$InitiateIOCTL**) performs control operations on the device. (See "Configuring RTAM" earlier in this chapter.)

In addition to the data, three function bits are sent to the device and three status bits are returned from the device. Other DR11 control information may also be sent for each request.

I/O Management

FS\$SetBlockIOparms sets the values for the function bits and other out-bound DR11 control information. This information is sent to PIAM in each Read, Write, or I/O control request. After each request, the device status, including the status bits, is returned from PIAM and stored in the file control block. To retrieve the device status, the application can call FS\$ReturnDeviceStatus .

The request, which includes control information, the buffer address, and length of the data to be moved is sent to PIAM. PIAM makes sure that the pages are set up for I/O, and forwards the request to the controller. The controller can then access the application's memory directly. However, when the amount of data is small, the data may be sent in the message along with the request or reply (as described in "Transferring Data" earlier in this chapter).

I/O Management

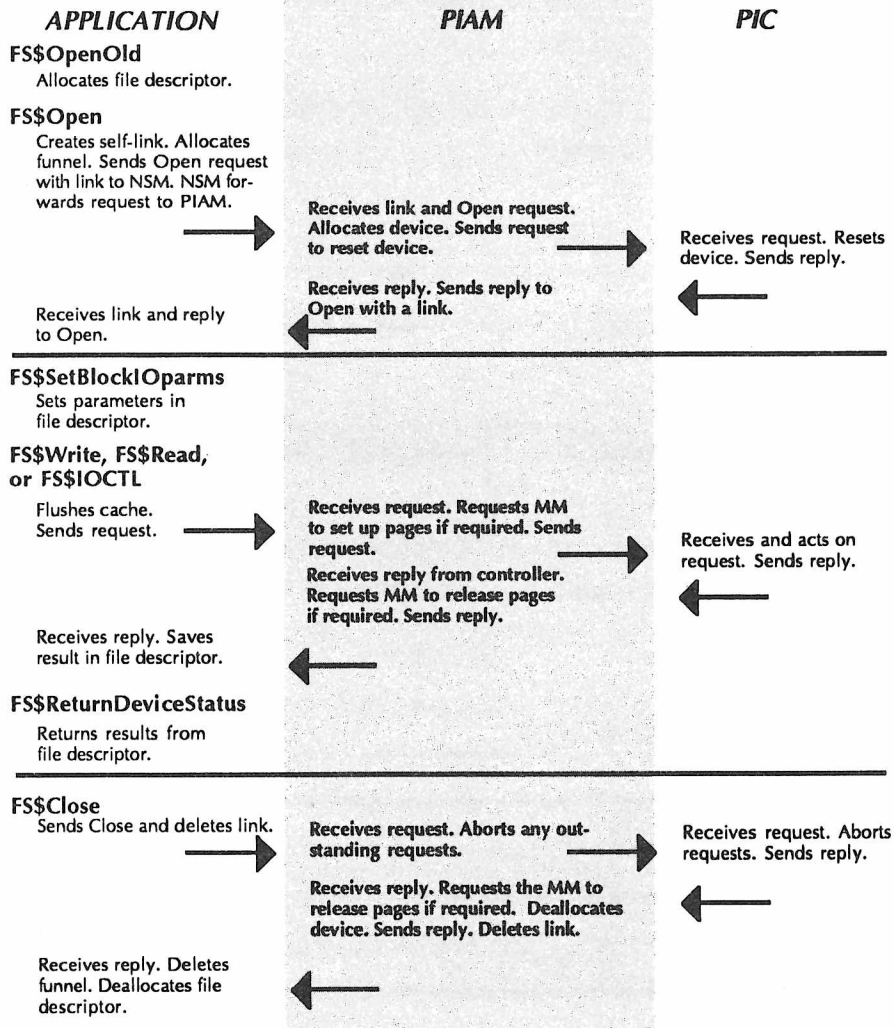


Figure 6-5. The Application, PIAM, and the PIC

I/O Management

PASCAL EXAMPLE FOR THE PIC

The following is a Pascal test program for the PIC subsystem. The name of the program is DR11Test.

DR11Test(output);

```
(***** *)
(*) *)
(*) This is a test program for the PIC subsystem. It requires two PIC (*)
(*) interfaces, both of which must be in Link mode. The inter- (*)
(*) face must be attached with the wrap cable and configured (*)
(*) by the system administrator. Here are examples of the con- (*)
(*) figuration commands: (*)
(*) *)
(*) ConfigController PIC1 type=PIC (*)
(*) *)
(*) ConfigPICdevice dr11a controller=PIC1 & (*)
(*) address=10 type=vaxLink (*)
(*) *)
(*) ConfigPICdevice dr11b controller=PIC1 (*)
(*) address=11 type=vaxLink (*)
(*) *)
(*) You must run two copies of the program, one in Master (*)
(*) mode and the other in Slave mode. (Slave mode is a switch.) (*)
(*) To set Slave mode, enter +slave or -master. To set Master (*)
(*) mode, enter -slave or +master. Be sure to give the interface (*)
(*) name, which should be identical to the name assigned when . (*)
(*) the device was configured. (*)
(*) *)
(*) You can also specify a verbose switch ( +verbose or -quiet). (*)
(*) If you specify verbose, a line is written to $stdout for each (*)
(*) transfer. If you specify -verbose (or +quiet), only errors are (*)
(*) reported. Here is an example of this program: (*)
(*) *)
(*) run 'dr11Test /dev/dr11b +slave' +detached (*)
(*) DR11Test /dev/dr11a +master (*)
(***** *)
```

```

%include '/embos/include/$include.p';
%include '/embos/include/FS$include.p';
%include '/embos/include/OS$include.p';

{*****}
{  Send the data in Link mode via the PIC. The first write (with  }
{  no data) waits until the read is ready. The second write sends }
{  the data.  }
{*****}
}

procedure writeDR11 (
  FD : $FDtype;
  bufferAddress : $addressType;
  bufferSize : integer);
begin
  FS$SetBlockIOparms (FD,
    FS$PICuseFunctionBits + {interrupt}
    FS$PICpulse + {other computer}
    FS$PICwaitForAttention, {and wait}
    2, {transmit + F2}
    bufferSize, {ODR = size}
    0, 0, 0, 0, 0 );
  FS$Write ( FD, 0, 0, true); {no data}

  if $status = $OKstatus then
    begin
      FS$SetBlockIOparms (FD,
        FS$PICuseFunctionBits + {cycle}
        FS$PICcycle, {to start}
        0, {transmit}
        0, {ODR}
        0, 0, 0, 0, 0 );

      FS$Write( FD, bufferAddress, buffersize, true );
    end;
  end; {procedure writeDR11}

```

I/O Management

```
{*****}
{  Receive data in Link mode via the PIC. The first read (with no  }
{  data) waits for a Write. The IDR (in the device status word)    }
{  contains the number of bytes the writing process is sending.     }
{  The second read signals the writing process and then accepts     }
{  the data.                                                        }
{*****}
}
```

```
procedure readDR11 (
  FD      :  $FDtype;
  bufferAddress      :  $addressType;
  bufferSize        :  integer;
  var transferSize   :  integer );
begin
  FS$SetBlockIOparms      (FD,
    FS$PICuseFunctionBits + {wait for}
    FS$PICwaitForAttention, {attention}
    1,                      {receive}
    bufferSize,             {ODR = size}
    0, 0, 0, 0, 0 );
  FS$Read(FD, 0, 0, transferSize);
  if $status = $OKstatus then
    begin
      bufferSize := min( bufferSize,
        os$bitWiseAnd(FS$ReturnDeviceStatus(FD), 65535);
      FS$SetBlockIOparms (FD,
        FS$PICuseFunctionBits + {interrupt}
        FS$PICpulse,           {other computer}
        3,                     {receive + F2}
        bufferSize,            {ODR=size}
        0, 0, 0, 0, 0 );
      FS$Read(FD, bufferAddress, bufferSize,
        transferSize);
    end;
  end;
end;      {procedure readDR11}
```

I/O Management

```
{*****}
{  Open a channel to the DR11 and initialize the device.  }
{*****}

procedure openDR11 (
    name      :   string;
    var chan   :   $FDtype );
begin
    if FS$OpenOld(chan, name, FS$ReadWriteDataAccess)
        = $OKstatus then
        FS$Open(chan);
end;

{*****}
{  Outer block  }
{*****}

const
    IntBufferSize = 1024;
    ByteBufferSize = IntBufferSize*4;
var
    i                :   integer;
    transferSize     :   integer;
    deviceName       :   string;
    fd               :   $FDtype;
    buffer           :   array [1..IntBufferSize] of integer;
    compareBuffer    :   array [1..IntBufferSize] of integer;
    verbose          :   boolean;
    size             :   integer;
    errorFlag        :   boolean;

BEGIN

{ Get command line parameters. }

parm ('deviceName +required');
parm ('slave antonym=master +switch default=-');
parm ('verbose antonym=quiet +switch default=+');
$StrArg('deviceName', deviceName);
verbose := $Switch('verbose');
```

I/O Management

```
if $Switch('slave') then
  { In Slave mode, echo all data. }
  begin
    { Open the device. }
    if verbose then
      writeln('<slave> opening device ', deviceName, '.');
    openDR11 (deviceName, FD);
    if $Status <> $OKstatus then $ErrorExit;

    { Declare I/O buffer. This is not required, but it does improve      }
    { performance. The I/O buffer is frozen in real memory.              }

    FS$SetupIObuffer(FD, adrord(buffer), ByteBufferSize);
    if $Status <> $OKstatus then $ErrorExit;

    { Echo data forever... }

    while true do
      begin
        readDR11 ( FD, adrord(buffer), ByteBufferSize,
          transferSize );
        if $Status <> $OKstatus then $ErrorExit;
        if verbose then
          writeln('<slave> writing ', transferSize:1,
            ' bytes of data. ');
        writeDR11 (FD, adrord(buffer), transferSize );
        if $Status <> $OKstatus then $ErrorExit;
        end;
      end
    else
      { In Master mode, generate test data, transfer, and compare. }
      begin
        errorFlag      := false;
        { Open the device. }
        if verbose then
          writeln('<master> opening device ', deviceName, '.');
        openDR11 (deviceName, FD);
        if $Status <> $OKstatus then $ErrorExit;
```


I/O Management

```
{  Declare the I/O buffers. This is not required, but it does      }
{  improve performance. The I/O buffers are frozen in real      }
{  memory.                                                        }

FS$SetupIObuffer(FD, adrord(buffer), ByteBufferSize);
if $Status <> $OKstatus then $ErrorExit;
    FS$SetupIObuffer(FD, adrord(compareBuffer),
        ByteBufferSize);
if $Status <> $OKstatus then $ErrorExit;
{ Initialize the buffer. }
for i                := 1 to IntBufferSize do
    buffer[i]        := i;

{ Loop, increasing the size of the requests, compares data.      }

for size := 1 to IntBufferSize do
    begin
        if verbose then
            writeln('<master> writing ', size*4:1,
                ' bytes of data. ');
        writeDR11 (FD, adrord(buffer[IntBufferSize-size+1]),
            size*4 );
        if $Status <> $OKstatus then $ErrorExit;

        readDR11 (FD, adrord(compareBuffer), ByteBufferSize,
            transferSize);
        if $Status <> $OKstatus then $ErrorExit;

        if transferSize <> size*4 then
            begin
                writeln('Transfer size compare Error.
                    Expected ', size*4:1, 'bytes, Received ',
                    transferSize, 'bytes. ');
                errorFlag := true;
            end;
```

I/O Management

```
for i      := 1 to size do
  if buffer[IntBufferSize-size+i] <>
    compareBuffer[i] then
    begin
      writeln('Buffer compare Error.
        Buffer[' , IntBufferSize-size+i : 1,
        ']' = ' , buffer[i] : 1,
        ' compareBuffer[' , i : 1, ']' = ' ,
        compareBuffer[i] : 1);
      errorFlag := true;
    end;
  end;      {for loop}
if errorFlag then $ErrorExit;
end;        {master}
END.
```

This chapter summarized the major elements of the System 6400's I/O system and described how to gain access to a device and transfer data. A FORTRAN example for the Realtime Access Manager and a Pascal example for the Parallel Interface Access Manager were also presented. The next chapter describes parallel processing on the System 6400.

GLOSSARY

access manager. A system process that controls the allocation and sharing of a device under Embos or EMS, and manages the flow of input and output to the device. Access managers provide a uniform interface for operations such as Open, Close, Read, and Write through the File System Intrinsic.

active channel. A channel with messages waiting to be received.

active list. A priority-ordered list of the processes ready to be executed on each CPU. The highest priority processes on the Active List are assigned by the Register Set Manager to register sets.

BIQ. Bus Information Quantum. BIQs are composed of small packets of information that are transmitted indivisibly between two functional units via the Gigabus. They provide the means for communicating between functional units connected to the Gigabus. BIQs are also used to send messages between hardware and software processes.

cache. High-speed memory managed with a two-way set associative mechanism that increases the average speed of access to main memory. Each CPU has a cache memory of 16 Kbytes (for the 6410 CPU) or 64 Kbytes (for the 6420 CPU) divided into blocks of 32 bytes.

cache flush. An operation in which a process writes the data in its cache to main memory.

channel. A method of grouping message funnels according to priorities. The number of a channel determines its local priority, with 1 being the highest and 15 (the default channel) the lowest.

Communications Supervisor (CS). The CS manages the Parallel Interface controllers, VME-based controllers, Ethernet controllers, and the other communication controllers on the system. Its main function is to manage controller configuration (including creating and deleting access managers), and downloading controller-code.

Glossary

CPU class. A CPU class is a set of one or more CPUs grouped together to bring scheduling and process migration under strict control.

data. Data is 64 bits in length when in registers, and is of variable length when in memory. Bits in data fields are numbered from left to right, high-order to low-order, with the most significant bit (or byte) as bit (or byte) zero.

datapool. A collection of variables that are shared among cooperating processes and exist in the private address space of each of the cooperating processes.

datapool descriptor. The symbol table contained in each datapool, from which one can find out the name, type, and location of all the variables in the datapool.

Elcon. The ELXSI system configurator program. Elcon takes three elements as input: a description of the hardware configuration, a description of the software processes that make up the System Foundation, and the configuration profile. Its output consists of a System image and an initial bootstrap file that resides on the Service Processor (SVP).

Ethernet controller. An integrated 68000-based microcomputer that provides the physical interface to an Ethernet network. The controller's major components are the Dynamic Front End (DFE) sub-bus interface, the Motorola 68000 microprocessor, dynamic RAMs for buffering code and data, and the Local Area Network Controller for Ethernet (LANCE).

execution class. The System Foundation divides the 256 global priorities into four execution classes. The highest of these is realtime priority, which is the execution class used by most System Foundation processes. Then in descending order, the execution classes are time-sharing, batch, and background.

File System Intrinsics. System routines in Embos and (in modified form) EMS that provide a uniform programmatic interface for accessing data in both disk files and devices.

funnel. The receptor into which a process sends a message.

Gigabus. A high-speed, synchronous, 64-bit system bus with an effective bandwidth of 320 Mbytes per second. All functional units of the system communicate via this bus. It accommodates multiple CPUs, memory modules, I/O Processors, and a Service Processor.

gigabyte. One billion bytes.

High-Speed Device (HSD). A general purpose 32-bit interface that serves as an HSD master and operates in either Normal or Interbus Link mode. Software on the controller implements all the driver functions.

Input/Output Processor (IOP). Allocates transfer channels and manages the message system for I/O processes. It is the interface to the Gigabus and manages access to main memory from the device controllers. Two sub-buses are connected to each IOP, each of which can support up to sixteen device controllers.

I/O Page Map (IOPM). The data structure the IOP uses to obtain the frame address of a particular page in main memory and determine whether a device controller has rights to access the page. The I/O Page Map has an entry for each page of physical memory.

I/O Sub-bus. A 16-bit wide, four-way interleaved synchronous bus. It has an aggregate data bandwidth of 8 Mbytes per second.

Lifeline Interrupt Handler (LIH). A system process that performs basic functions of process control. The LIH consists of a special message system funnel that is set at the highest priority and an interrupt routine associated with that funnel.

link. A message system mechanism that provides a one-way communication path from a sending process to a receiving process. Each link points into exactly one funnel, but a single funnel can have many links pointing into it.

link code. An arbitrary number defined by the receiving process when it creates the link. A link code identifies the message and is associated with each link.

mapped-file access. Data on disk is mapped into the application's virtual address space rather than passed through a separate process. Thus,

Glossary

by merely reading and writing memory locations, the file itself is read and written to.

Memory Manager. A system process that manages the virtual memory system.

message. The primary means of communicating among system processes. A message consists of a message parameter block, followed by an arbitrary sequence of data, with a data length of up to 888 bytes.

NameSpace Manager (NSM). A system process that enforces Embos, EMS, and System Foundation security and controls access to named objects in the system, such as disk files, devices, or other processes. There are several types of entries in the NameSpace, such as directories, disk files, aliases, rendezvous, and server rendezvous.

page. 2048 bytes.

page fault. An attempt to access a virtual page that is not in real memory.

page map. A data structure used by the hardware that defines where each of a process's pages of virtual memory is located, either in physical memory or on the disk. Each page map entry describes one page of virtual memory, including its location and access rights. For each subspace in virtual memory, there is a separate page map.

Parallel Interface Access Manager (PIAM). A block I/O access manager for devices attached to the Parallel Interface Controller and a generic access manager for PIC devices.

Physical-Virtual Address (PVA). An index to page map entries (created by the Memory Manager) that is received by a software process. This index is used for subsequent data transfer operations.

primitive messages (PMSGs). An alternate form of the general message system that can be employed for special circumstances in which both the speed of message delivery is critical and the amount of data to be transferred is very small.

private space. Virtual address space used for the private data and code of all processes. Each process has two gigabytes of separate private space that cannot be addressed by any other process (except when it is explicitly shared).

process. An executing program, including its code, data, registers, and allocated system resources, such as memory. It is the basic building block of the system.

process context. A collection of information about each process maintained in the Process Control Block (PCB). This information consists of a unique process ID that is recognizable across all CPUs, a CPU on which to execute initially, a priority, virtual memory space, and several control blocks that manage the process and its messages.

process fork. A system service that produces a new process by duplicating a process's address space (that is, its code and *unshared* data). The child process receives a "snapshot" copy of the parent's address space, duplicating its access rights, cachability, and current page contents. Both the parent and the child process execute concurrently.

process ID. A number that identifies each active process. A process ID is unique across all the CPUs and devices on a System 6400.

Process Manager (PM). The system process that handles process skeletons (for Embos) and directs the creation of a new process (when a process skeleton cannot be used). It starts with information stored in the program file that is being forked or spawned and allocates the system resources necessary to create a process suitable for executing that program.

process migration. A system service that moves processes from one CPU to another CPU in order to balance the workload across CPUs. It is the mechanism by which the system load can be dynamically adjusted.

process skeleton. A set of all the necessary data structures required to create a process.

Glossary

public space. Virtual address space that is common to all processes and can be addressed by all processes in the system. A particular byte in the public space has the same address in every process. Public space is read-only.

Realtime Access Manager (RTAM). A system service that provides direct access to most types of device controllers, including VME-based controllers, Parallel Interface controllers, and Ethernet controllers. Realtime applications have complete control of the interface to the device controller. RTAM provides some operating system services, but it does not interface directly with the controller.

register set. Data structure maintained by the CPU that contains or points to all the information needed to execute a process. There are a total of sixteen register sets per CPU: one is reserved for the CPU itself and another for the Register Set Manager.

Register Set Manager (RSM). A System Foundation process that works closely with the CPU microcode to control the priority and scheduling of processes on each CPU. The RSM also works closely with the Process Manager (PM) to perform the process migration and CPU Failsoft functions. There is one RSM process per CPU.

self link. A link pointing into a funnel that belongs to the same process. Self links are an important part of setting up process-to-process communication paths.

shadow file. A temporary disk file to which pages of virtual memory can be written when (or if) the Memory Manager takes these pages out of memory. Each process has a shadow file.

server rendezvous. An entry in the NameSpace that allows applications to establish communication with a single server process (such as an access manager).

Service Processor (SVP). An independent microcomputer system that starts up the primary system, monitors the hardware, logs error conditions, and diagnoses failed hardware components. The SVP runs the system diagnostics, performs bring-up tests when the system is initialized, loads the instruction microcode, and then bootstraps the operating system. It is attached directly to the Gigabus.

simple rendezvous. An entry in the NameSpace that establishes direct communication between two processes.

spawn. A method for creating a new process in which all of the essential components of a process must first be identified and brought together into an executable form.

stale data. A situation in which data is changed in a CPU's cache but not in memory (or changed in main memory but not in a CPU's cache).

subspace. Any one of four sections of the four gigabyte virtual address space, commonly referred to as P0, P1, P2, and P3. P0 and P1 are private space, P2 is public space, and P3 is reserved.

supervisor. A system process responsible for the configuration, initialization, and error handling of device controllers. Supervisors are part of the bootstrap image created by Elcon.

SVP operating system (SVPOS). A general purpose operating system that is stored on the SVP hard disk and loaded into SVP Memory. SVPOS controls diagnostic testing and interrogation of each hardware functional unit.

system logfile. A file in which any abnormal conditions detected by the SVP while monitoring or diagnosing the system are noted. In addition, System Foundation processes log error conditions and other system events into the same logfile.

system message buffer. A separate area of memory reserved for the message system.

System Foundation. A network of independent processes, each of which controls one or more system resources, such as CPUs, memory, disk access, other I/O access, and so forth.

Glossary

System Intrinsics. A large set of library routines that provide access to the System Foundation services and Embos library functions such as sorting, mathematical functions, pattern matching, and so forth.

transfer channel. The physical conduit that gives a device controller the ability to transfer data from the controller to memory. The IOP assigns four transfer channels per I/O sub-bus.

UNIX PID. The process ID under UNIX. This number is unique only within a particular UNIX system. UNIX refers to the system-wide process ID as an *EPID* (ELXSI Process ID).

VME Front End (VFE). Connects the sub-bus to an industry standard VME bus. It performs the functions of the Dynamic Front Ends (DFE) on standard System 6400 device controllers.

write-back cache. A cache scheme in which Writes to cache are *not* written to main memory at that time. The changed data is written to memory under two conditions: when the cache block is reused for new data and when the cache block is explicitly flushed.

write-through cache. A cache scheme in which every Write to cache memory is automatically written to main memory as well.

INDEX

- \$all 3-22
- \$any 3-22
- access control list 6-12
- access manager
 - customized 2-20, 6-2
 - purpose of 2-19
- active list 3-15
- binary lock 7-11
- Binder 3-10, 4-3
- Bound file 3-10
 - Info Page 3-10
- BIQ 2-9
- bulk I/O 2-6
- cache 4-16
 - block 2-1
 - bus-following 4-18
 - flush 4-17
 - noncachable data 4-19
 - stale data 4-17
 - on System 6400 4-18
 - TLB 2-10
 - write-back 2-10
 - write-through 4-18
- channel
 - active 5-19
 - local priorities 5-3
 - priority 3-19
- child processes
 - CPU class of 7-2
- Communications Supervisor 6-9
- components, system 2-8
- context switching 1-3
- controller
 - code 2-18, 2-22
 - configuring 2-21
 - as device drivers 6-1
 - identified by 6-12
 - loading 2-22
 - location 6-12
 - post-processor 6-12
 - as a process 6-1
 - transferring data 6-21
 - types 2-14
- CPU class 3-22
 - \$all 3-22
 - \$any 3-22
 - of child process 7-2
 - cooperative 3-27
 - restricting users 3-24
 - restrictions on 3-23
- CPU Failsoft 3-15, 3-29
- critical regions 2-3
- dangling link 3-8
- datapool 4-28
 - assembler 4-28
 - case sensitivity of 4-34
 - declaring variables 4-30
 - descriptor 4-28
 - initializing data 4-34
 - intrinsic 4-29
 - namescoping 4-34
 - naming convention 4-35

Index

- datapool (cont.)
 - parts of 4-28
 - section names 4-35
 - sharing memory 4-32
- debugging
 - and subspace address 4-3
- DEC DR11 4
- device
 - access manager and 6-16
 - access to 2-22, 6-16 to 6-17
 - direct 2-23
 - via access manager 2-22
 - closing 6-17
 - configuration 6-13 to 6-14
 - direct access 6-17
 - realtime 6-15
 - restricting access 6-14
- device controllers
 - as software processes 2-3
- device driver, ENIX 2-20
- driver functions 2-18
- ECC 2-16
- Elcon 2-2, 2-18, 2-21
- EMS 3-7
- EPID 3-6
- EPROM 2-16
- Ethernet Controller 6-6
- executable file 3-9
- execution class
 - background 3-17
 - batch 3-17
 - realtime 3-17, 3-18
 - timesharing 3-17
- exception dispatcher 5-32
- explicit residency 3-28
- File System Intrinsic 3-4
- fork-shared memory 4-24, 4-25, 5-30
- freeze process 4-15
- funnels
 - and channels 5-3
 - create 5-3
 - delete 5-3
 - disable 5-4
 - funnel table 5-2
 - standard 5-32
- Gigabus 1-1, 2-1, 2-9
- global priority 3-17
- HSD 6-7
- Info Page 4-3
- instruction set 2-1, 2-11
 - data types 2-11
- Interactive Access Manager 2-7
- interprocess communication 5-27 to 5-44
 - message system 5-29
 - parent/child 5-35
 - rendezvous 5-29
 - shared memory 5-30
 - spawn or fork 5-28
- interrupt routine 5-43 to 5-44
- interval timer 1-3, 3-31
- I/O
 - bandwidth 1-20
 - bulk 2-6
 - Comm Supervisor 6-9
 - configure device 6-13 to 6-15
 - Ethernet controller 6-6
 - file security 6-11

- I/O (cont.)
 - HSD 6-7
 - I/O page map 6-21
 - IOP 6-2
 - memory 6-19 to 6-20
 - Memory Manager 6-9
 - Namespace Manager 6-11
 - page map 6-3, 6-20
 - in parallel 2-20, 6-1
 - PIAM 6-11, 6-28 to 6-30
 - PIC 6-4 to 6-5
 - PVA 6-9
 - RTAM 6-9, 6-22 to 6-27
 - server rendezvous 6-11
 - setting up 6-21
 - sub-bus 2-14, 6-1 to 6-3
 - system elements 6-2
 - To-hardware message 6-12
 - transfer channel 6-3
 - transferring data 6-18 to 6-21
 - VFE 6-7
- I/O Page Map 6-3, 6-20
- I/O Sub-bus 2-14, 6-3
- Input/Output Processor 2-12
 - to 2-14, 6-2
- job 3-6
 - batch 3-7
 - timesharing 3-7
- Keyed File Access Manager 2-7
- labeled common 4-22
- Lifeline Interrupt Handler 3-12
- link
 - create 5-4
- link (cont.)
 - delete 5-5
 - and funnel 5-2
 - link table 5-2
 - notification attributes 5-10
 - to other processes 5-32
 - rights of 5-10
 - self-link 5-2
 - standard 5-32
- load factor 3-21, 3-28
- local priority 3-17
- lock, use of 7-11
 - binary 7-11
- mapped file access 4-48
- max attached limit 5-22
- max in-transit limit 5-22
- memory
 - Embos 4-6
 - EMS 4-6
 - fork-shared 4-24, 4-25
 - freeze 4-15
 - heap 4-3
 - map space 4-3
 - private shared 4-22, 5-30
 - public shared 4-22, 5-30
 - sharing 4-20
 - system 2-11
 - UNIX 4-7
 - virtual 4-2 to 4-7
- Memory Manager 6-10
- message
 - data length 5-5
 - deleting 5-18
 - delivered 5-19
 - formats 5-38
 - header 5-7

Index

- message (cont.)
 - notification 5-10
 - parts of 5-5
 - receiving 5-11 to 5-17
 - sending 5-6
 - To-hardware 6-12
 - work-in-progress entry 5-41
- message system
 - channels 2-4
 - constraints 5-22
 - Embos C 5-48
 - FORTRAN 5-45
 - funnels 2-4
 - and interrupts 5-20
 - I/O 2-6
 - link code 2-5
 - links 2-4
 - Pascal 5-44
 - performance 2-5, 5-23
 - and priorities 5-18 to 5-22
 - realtime response 5-24
 - rendezvous 5-29
 - Send routines 5-8
 - transfer rate 5-23
 - UNIX C 5-50
- migration. See **process**.
- modular expansion 1-1
- Namespace
 - entries in 6-11
 - Manager 6-11
- noncachable page 4-20
- notification message 5-10
 - delete link 5-10
- P0, P1. See **private space**.
- P2. See **public space**.
- Page 0 2-12
- page fault 3-15, 4-13
- page map
 - definition of 4-8
 - entry 4-8, 6-20
 - level-1 4-8
 - level-2 4-8
 - level-3 4-8
 - root 4-10
- Parallel Interface Access Manager 6-11, 6-28 to 6-30
- Parallel Interface Controller 6-4 to 6-5
 - Pascal example for 6-32 to 6-38
- parallel processing
 - awaken process 7-12
 - binary lock 7-11
 - cachability 7-4
 - create child process 7-11
 - creating a semaphore 7-9
 - debugging tools 7-16 to 7-17
 - flush memory 7-15
 - flushes 7-5
 - guidelines 7-18 to 7-20
 - lock a process 7-14
 - locks 7-20
 - noncachable data 7-4
 - process descriptor 7-12
 - process names 7-2
 - record pointer 7-12
 - semaphores 7-19
 - sharable and cachable data 7-5
 - shared data 7-2, 7-18
 - sharing memory 7-3 to 7-9

- parallel processing (cont.)
 - task number 7-15
 - unlock a process 7-14
 - variables 7-5 to 7-9
- Physical-Virtual Address 6-9
- PMSG 5-24 to 5-27
 - delivery 5-25
 - receptor 5-25
 - size of 5-24
- primitive message. See **PMSG**.
- priority
 - global 5-18
 - local 5-18
 - interrupts 5-22
 - lowering 5-20
 - map 5-19
 - realtime 3-17
 - modifying 3-18
- private space 2-12, 4-2
- private-shared memory 4-22
- process
 - adjusting priority 3-17
 - blocking conditions 3-15
 - channels 3-18
 - child 3-11
 - context 2-2
 - definition of 2-2
 - execution rule 3-15
 - fork 3-11, 4-24
 - advantage of 3-11
 - freeze 4-15
 - ID 3-6
 - lock 7-14
 - lock in register set 3-16
- process (cont.)
 - migration 3-15, 3-21
 - and load factor 3-22
 - parent 3-10
 - realtime 3-5, 3-16 to 3-18, 4-15, 4-50, 6-11, 6-22
 - Shell 3-9
 - skeleton 3-8
 - spawn 3-10
 - super-realtime 3-18, 3-19, 3-20
 - terminate 3-13
 - Hardkill 3-14
 - Softkill 3-14
 - unlock 7-14
- Process Control Block 2-2
- Process Location Table 3-6
- Process Manager 3-9
- program list 6-12
- public space 2-12, 4-2
- PVA 6-9
- quantum fault 3-15
 - and timeslice 3-16
- raw-mapped files 4-50
- realtime
 - modifying priority 3-18
 - priority range 3-18
 - super-realtime 3-18
- Realtime Access Manager 6-11, 6-22 to 6-27
- received message
 - in two buffers 5-12
- Register Set Manager 2-9, 3-9, 3-15
- register sets 3-15

Index

- rendezvous
 - types of 5-30
- resources, expanding 1-5
- root page map 4-10
- self-link 5-2
- semaphore
 - versus lock 7-10
- server 5-36
 - multi-threaded 5-36, 5-41
- server rendezvous 5-29, 5-34, 6-11
- Service Processor. See **SVP**.
- shadow file 3-8, 3-9, 4-14
- sharing memory 4-20, 4-21
 - with cached pages 4-21
 - datapools 4-32
 - MM\$ intrinsics 4-23
 - private 4-22
 - public 4-21
 - \$ShareMemory 4-22
 - under UNIX 4-26
- Shell 3-9
- simple rendezvous 5-29
- spawn 3-10
- stale data 4-17
- super-realtime 3-18
 - launching 3-20
- supervisor 2-18
- SVP 1-2, 2-2, 2-15
 - components 2-15
 - controller 2-16
 - downloading from 6-12
 - functions 2-15
 - interfaces 2-16
- SVP (cont.)
 - logfile 2-15, 3-4
 - EMS 3-5
 - UNIX 3-5
 - memory 2-16
 - operating system 2-15, 2-17
- SVPOS 2-17
- synchronization
 - among CPUs 1-3
 - among processes 7-13, 7-15, 7-35
- System Foundation
 - message system 3-1
 - realtime processes 3-5
 - UNIX file system 3-3
 - virtual machine 3-2
- System Intrinsics 3-3
- system message buffers 5-6
- timer services 3-29, 3-30
- Timesharing/Batch Monitor 3-7
- timeslice 3-15
- transfer channel 6-2, 6-3
- transferring data 2-24, 6-18 to 6-21
 - read 6-19
 - via DMA 2-24
 - via message system 2-24
 - write 6-18
- UNIX 3-2
 - branch tables 4-5
 - Exec 3-11
 - file system 3-2
 - fork 3-11
 - Init 4-14

UNIX (cont.)

lifeline handler base 4-5

sharing memory 4-26

and skeletons 3-8

virtual memory 4-4

version ID 3-6

VFE 6-7

virtual address 4-9

virtual memory. See **memory**.

write-through cache 4-18

